IAN SINCLAIR

WORKING WITH MSX BASIC

271

Working with MSX BASIC

Working with MSX BASIC

Ian Sinclair

COLLINS
8 Grafton Street, London W1

Collins Professional and Technical Books
William Collins Sons & Co. Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Collins Professional and Technical Books 1985

Distributed in the United States of America
by Sheridan House, Inc.

British Library Cataloguing in Publication Data
Sinclair, Ian R.
Working with MSX BASIC.
1. Basic (Computer program language)
2. Microcomputers—Programming
I. Title
001.64'24 QA76.73.B3

ISBN 0-00-383103-5

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

MSX manufacturers will also offer cassette recorders, though if you have a recorder already it can probably be used. Many of the MSX manuals say very little about tuning a TV receiver to the computer's signals, or about checking and adjusting cassette recorders, and so these topics have been dealt with in Appendix A.

Another important point about MSX is that the system can be expanded; moreover you are not compelled to buy products from just one manufacturer. Many computer manufacturers in the past designed their machines so that it was almost impossible to use additional equipment from any other supplier. This meant that when you decided to scrap the computer, you had to scrap everything else as well. You no longer have to accept this situation, because MSX uses standardised connections and signals. Any computing equipment that you buy for one MSX machine should be usable on any other MSX machine. In addition, if you already have a printer which is fitted with the standard Centronics connection you will be able to use it with your MSX computer; all you will need is a connecting cable.

If you have used another type of computer, perhaps at school, at work, or at home, then you will readily understand the advantage of the MSX system from reading this book. In particular, you will appreciate the ability to carry out precise arithmetic, and to have easy control over sound and graphics. As your programming knowledge advances, you will find that MSX has even more to offer, particularly if you are interested in lists and in filing data. For beginner or expert alike, MSX has a bright future.

I would like to emphasise that this book was written while I was *using* a Toshiba MSX computer, and that the listing of programs in this book were obtained from an Epson printer that was connected to the MSX computer. This might seem to be an unnecessary claim, but many books still appear in which the program listings have been retyped, with errors appearing in many of them. Every program which appears in this book, and every example of programming commands, has been tested on the MSX computer which I have here in front of me. Nothing has been copied untested from the manual, and where a command has operated in a way that is not obvious from the manual description I have pointed out the difference. One price that I have had to pay for this has been the disappearance of the £ signs from several listings, notably in Figs 2.18 to 2.20. This is because the pound character which appears on the computer screen does not appear on my printer, but I have noted in the captions where the pound sign should be typed in. All of the screen displays which I have described were obtained on a Fidelity TV/monitor, which was used as a normal TV receiver.

As always, I am greatly indebted to many people who have made this book possible. The machine was provided by Toshiba (UK) Ltd., and Richard Miles of Collins Professional & Technical Books worked tirelessly to ensure that I had this MSX computer on my desk as soon as possible. I

Contents

Preface

MSX is the name for a set of standards to which many computers are being constructed at present, and to which many more will be constructed in the future. Until the advent of MSX, a tape which had been recorded by one make of machine could not be used on any other machine. Worse still, the programmer of one make of machine would find it quite difficult to switch to another machine without relearning the language. MSX has at long last ended this ridiculous situation, so that anyone who buys an MSX computer will be able to exchange tapes and program ideas with anybody else using MSX. The manufacturers may be different, but the machines work in exactly the same way. An additional bonus for the retailers is that many of the MSX machines are manufactured by companies who are respected and trusted, with a long history of success in radio, TV and hi-fi.

If you have bought this book as a guide to MSX before buying, a few hints on machines may be helpful. Though all of the machines will run the same tapes, and be programmed in the same way, they are not identical. Some, for example, have better keyboards than others. For anyone who does any more with a computer than play games, the provision of a good keyboard is very important. Some machines can be connected to many more devices than others. The business user of a computer will want to connect up a printer and a disk drive, and this is provided for in *all* MSX machines. If you want to use your computer to control a music synthesiser or to work with pictures from a video camera, however, you may find that some machines are better suited to this than others. Very often, you will find that the items to which your MSX computer can be connected reflect the special interests of the manufacturer.

To make up a computer system you need the MSX computer, a cassette recorder, and a TV receiver or monitor. All MSX machines provide colour signals to colour TV receivers, and sound signals also. Similarly, all MSX machines can be used with monitors. A monitor is a form of TV which has been designed to take signals directly from a computer or a video recorder, rather than from an aerial. It gives a much clearer picture than you can ever get with a TV receiver. Many manufacturers of MSX computers will probably offer colour monitors in addition to the computers. Most of the owe a special debt of gratitude to Richard Yardley, of Spectravideo Ltd. He alone was able to get me a printer cable for the MSX computer at a time when there appeared to be only two such items in the country. Among many others at Collins Professional & Technical Books, Richard Miles, Janet Murphy and Sue Moore worked wonders with my manuscript, and the most efficient team of typesetters and printers that I know operated to produce the book in record-breaking time. I am sure that the result of all this work will be a book that will match the capabilities of your MSX computer.

Ian Sinclair

Chapter One
Where Do We Start?

Welcome to MSX computing. By this time, you don't have to be told how well-designed and constructed your new MSX computer is. What you probably want to know, however, is just how you can get more out of it. If you intend to use the computer as you use a washing machine, running only programs which are built-in or which you can buy, then the manual which comes with the machine will serve you well. But buying a computer and not programming it for yourself is like buying a Porsche and getting someone else to drive it. This book is all about how you can make your MSX computer do what *you* want, so that you can make full use of all these keys that you have paid for.

To start with, just what can the computer do for you? You may have bought it to play games, as many owners do. Games nowadays are designed by professionals, using much larger computers, so you can't really hope to come up with a new and earth-shattering game with a small machine. What you *can* do, though, is to use the machine for your own purposes. You might want to keep track of the members of your Rugby Club, or the purchases from your mail order catalogue. You might want to devise educational games for your children; games which are suited to their particular needs. You might want to run a local Camera Club, keep track of shares on the Stock Exchange, analyse what your car costs, check your income tax ... the list is endless. Whatever your own interest is, whether it's tracing your ancestors or cataloguing the works of a long-dead writer, you'll need a program.

A computer is a brainless machine, whatever anyone says. What makes it work is a set of instructions that we call a program. You can write such a program for yourself, so that it does what you want. Now if your program is useful to you, it might possibly be interesting to someone else. You might be the only person using the computer to keep the scores and records of the Little Tiddleworth Darts Club, but there could be a lot of other people who would like to try out your program for their own uses. That's where MSX comes in. If you have written a program for one MSX computer, and recorded it on tape or on disk, then it should be usable on any other MSX computer so long as there is enough memory in that machine to hold the program. MSX is a system that is used by many computer manufacturers for just this reason.

What is a computer, then? The short answer is that it's a machine which can organise information for you. Give it a set of names and addresses, and it will store them, arrange them into alphabetical order, produce the address for a name that you type, and so on. Give it a list of names and payments to a club, and it will give you a list of who paid how much and who still owes what. Give it a group of numbers in answer to questions about your earnings and expenses, and it will show you your correct tax code. Does this sound like a brainless machine? If it doesn't, that's because all of the organisation is done under the command of a program. The program is what we call the *software* of the machine. Without a suitable program, the computer can do nothing. The machine itself is the brainless bit, the clever part is the program. So how do *you* go about writing a program?

It isn't as difficult as you might think. The reason is that there has been a great deal of progress in this respect in the last twenty years. Many years ago computers were big machines which had to be programmed by skilled people who understood the number code system – called machine code – that the computer used. As time went on, programming languages were invented. A programming language is a way of instructing a computer by using command words which are English words. Even before micro-computers were dreamed of, there were several of these programming languages, but the most important one for us is BASIC. BASIC is short for Beginners All-purpose Symbolic Instruction Code. It was invented at Dartmouth College, USA, as a way of teaching computer programming. Since then, it has spread like wildfire to become the programming language that is used for each and every home computer. It's still a good language for beginners in the sense that it's easy to learn, but it has also developed into a language that gives you a great deal of control over your computer.

Your computer uses a special variety of BASIC which is called MSX BASIC. It's one of the most up to date varieties of BASIC, and yet it has been developed from one of the first versions of BASIC used in small computers; Microsoft BASIC. What that all boils down to is that MSX BASIC is easy to use, reliable, and yet capable of getting the best from your computer. Unlike the older varieties of BASIC, too, it will work on any MSX machine. Can you imagine a world in which your records wouldn't play on your neighbour's hi-fi system? Would you buy a cassette player if you knew that only one make of cassette could be used with it? That's the state that computing was in before MSX. Now for anyone who doesn't have an MSX machine, that's the way it still is, but for *you*, things are different. Your computer can use a tape or disk that was recorded by any other MSX computer. Anyone else who has an MSX computer can make use of one of your tapes or disks. That's the way it should have been in the first place, but it has taken MSX to do it. If you have never tuned a TV receiver, or adjusted a cassette recorder, there's help for you in Appendices A and B at the end of this book.

BASIC foundations

The BASIC language is built like any other language, with vocaculary and syntax. Vocabulary means the words that are used to make things happen. Syntax means the way that you have to use the words. Like any other language, BASIC has rules which have to be learned. What makes it so much easier for you to learn is that the words are mainly English words, and the rules are simple. Moreover, your computer will tell you when you have disobeyed the rules!

To learn about BASIC, then, you have to learn what a number of words mean. There are about one hundred of these words, and the meaning of most of them is obvious. In addition, you must learn the correct way to use each word. This is where you really come up against the 'brainless machine' bit. The computer can make use of a command which uses the correct word in the correct way. If you use the wrong word, spell the word wrongly, or use it in the wrong way, the machine can't do anything – so it stops. When it does this, a message will appear on the screen to tell you what has happened. This is called an *error-message*, and from what it says, you should be able to find what went wrong.

If that makes it sound simple, that's because it *is* simple, and that's why it needs to be learned. Suppose, for example, you had a foreign friend who understood only about one hundred words of English. How would you make him understand, using only the words he knows, that you want him to do something like arrange words in alphabetical order or find how many members of the dominoes club have paid their subscriptions? The answer is that you have to break up these complicated actions into a lot of simple steps. You then have to describe each step in the words that your hundred-word friend understands. There's one big difference here, though. A human who knows a hundred words of a language will pick up other words very quickly. A computer doesn't, so far, have this ability. Every instruction has to be made out of these important command words, the *keywords* as we call them. A program consists of an arrangement of these keywords in such a way that something useful is done. A program might be educational, amusing, for business or leisure, working with words or with numbers; whatever you want it to be. Whatever it is, though, it still has to make use of these keywords, because that's all that the machine can understand. You can't type instructions like 'Who ordered a dozen spob-tackers?' or 'What will my payments be if I pay back one fifth of the mortgage?'. These are questions for humans with human brains, not for computers. Computers have memory. They can store information and they can store instructions.

They can't puzzle out what a new instruction might mean, though, which is why we have to write programs.

Structure of BASIC

One of the command words in BASIC is used here to let you see what a BASIC program looks like. The word is LIST, and you use it by typing the word, then pressing the key that is marked RETURN or ENTER, or indicated with a large arrow which curls down and left. The RETURN/ENTER key is an important one. It's the 'do it now' key that makes a command work. Why do we need it? The brainless machine, you see, can't tell when you have finished typing, and it needs a signal to tell it that what's there is everything – get on with it. The RETURN/ENTER key provides that signal. If you have a BASIC program in the memory of the computer, then typing LIST and pressing the RETURN/ENTER key will make the program appear. Unless the program is very short, it will take up much more space than is available on the screen. As new instructions appear at the bottom of the screen, then, previous ones vanish from the top. It's just as if the screen was a frame, and the program on a long sheet of paper being wound through the frame. This action is called *scrolling*. Because of scrolling, what you normally end up with when you use LIST is the last few sets of program instructions. You can stop the scrolling at any stage by pressing the STOP key once. Pressing STOP a second time will restart the scrolling. If there is no program stored in the memory of the computer, then using LIST will produce nothing. Try it for yourself by loading one of the sample programs that came with your computer, and making it stop by pressing the CTRL and STOP keys together. You can now use this LIST action to see what BASIC commands look like.

Suppose, then, that you *are* looking at a program listing. What do you see on the screen? The answer is a set of numbered lines of instructions which use familiar looking words. These are lines of BASIC, and the numbering is there as a guide to the computer. Normally, the computer will work on the lines in order of increasing number. This way, the line numbers show in which order the computer must carry out the instructions. Another use for these line numbers, however, is *branching*. You can include in the program instructions which force the machine to work on another line. This allows the program to vary what the machine does. For example, you might want it to print from a list of names each one whose subscription is due. This means selecting names, not just printing the whole list. At some point in the program then, the machine will be instructed to print a name instead of just moving on to the next name. This is one type of branching action. Another one is *looping*. Looping means that the machine repeats a set of instructions, carrying them out over and over again instead of keeping to the strict number order of the lines. These types of action are very important in computing, and in BASIC, the line numbers are used in the commands which make the machine carry them out. The line numbers, then, are an essential part of BASIC. If you type a number, the machine will always take this to be a line number.

You can't use the computer in the way that you would use a calculator. You can't, for example, type:

```
26 + 42 =
```

as you would with a calculator, because the computer is programmed to take 26 as meaning a line number. Since it is *not* programmed to do any sort of arithmetic with line numbers, it can't make any sense of the + 42 = that follows, so it issues the 'Syntax error' message in the hope that you'll get it right next time. Once you understand that this is a simple-minded device and not a pocket genius, you start to find that programming becomes much easier!

One other thing that will not bother you much at the beginning is memory. The computer stores all the instructions for a program in its memory; it also needs to reserve some memory for making notes as it goes along, and for organising the display on the TV screen. Memory is measured in units that are called *bytes*. One byte is the amount of memory that is needed to store one instruction word, or one letter of any other words. Because one byte is a small amount of memory, we use the word *kilobyte* more often. One kilobyte (shortened to 1K) is 1024 bytes of memory. Many machines claim to have 64K of memory, and they do, but only a part of this can be used by you. Normally, if your machine releases 28K or so for you, there will be enough memory for any program that you are likely to write for yourself.

Summing it up so far, then, programming means writing instructions which are broken down into simple steps. These steps make use of a limited number of keywords, which must be used in the correct way and with the correct spelling. The keywords are, in turn, arranged in numbered lines, so that the computer can carry out the instructions in the correct order. From all this, you'll see that typing is an important part of all this activity, and so we'll look next at the keyboard on which you have to type all of these keywords and the other parts of instructions.

The keyboard

Since the keyboard is the way that we enter the instructions of a program into the computer, we need to take a close look at this vital component. Now just because all of the MSX machines will run the same programs and can be programmed in the same way, that doesn't mean that all the keyboards will be alike. Some machines will use good-quality keyboards with moving keys as you would expect in a typewriter. Others may use the flat-pad keys, or the calculator-style keys that can save the manufacturer so much money. If you are going to do a lot of programming, or if you intend to use the computer for anything other than games, a good keyboard is essential. More working computers are thrown away because of poor keyboards than for any other reason, and if, like me, you spend a lot of time at the keyboard, you'll understand why. When you begin computing, you may think that the keyboard doesn't matter, but as you move on, you can soon be reminded of how important a good keyboard is.

When you have the computer ready for programming, you will see on the screen a marker which is called the *cursor*. On most machines, this marker draws attention to itself by flashing. When you press a key which is labelled with a letter or number or anything else that can be printed, then the place where it will appear on the screen is marked by this cursor. If you press the A key, for example, you will see the letter a or A appear where the cursor *was*, and the cursor will move one space to the right. When the cursor gets to the right-hand side of the screen, you don't have to do anything different. Just press the next letter that you want, and you will see the letter appear; the cursor will move to the start of the next line, at the left-hand side. You do *not* have to press the RETURN or ENTER key to make this cursor movement happen. Note that the bottom line of the screen shows the set of keywords that can be obtained simply by pressing one of the F keys on the top row of the keyboard. Another set of words appears when you press the SHIFT key, and this shows what can be entered when the F keys are used along with the SHIFT key.

The style of keyboard can vary, and so also can the keys that you find on it. The main keys, which give the letters, numbers, and punctuation marks, are the same on all MSX keyboards. On many machines, these keys are in a lighter colour than the other keys. What can be different, however, is the other keys that surround these, and so we'll look at the other keys first. The most important one is the key which on some computers is marked RETURN, and on others is marked ENTER. A few computers use a symbol, which is a thick arrow that curls downwards and left. It's the same key, doing the same job, and is in the same place on any keyboard. It's placed so that you can hit it with the little finger of your right hand. The effect of this key is, as you know already, to signal to the computer that you have finished a command or a reply or a program line, and you now want the machine to do something about it. The word ENTER is found on some keyboards because this key will enter what you have typed into the memory. The word RETURN is used on others because the key is in the position of the carriage return key of a typewriter. Either way, it helps if this key is bigger than the others, providing it doesn't jam if you hit it off-centre.

After the RETURN/ENTER key, we can look at the others in any order we like, and the easiest way is to look first at the top row of keys – the row which contains a set of five keys that are marked with F1,F2... up to F10. These are called the *programmable function keys*, and the idea is that you use them to save yourself typing. For example, when you are entering a program, you may find that you have to use a keyword, or a combination of keywords, over and over again. Each of these special keys can be programmed to do this work, so that you need type the word(s) once only. From then on, you use the programmable function keys for the words that you have programmed into them. For example, if you find that you need to keep using the word PRINT, then you could program this into F10, and hit F10 each time you wanted the word PRINT to appear on the screen. The words that are assigned to these keys will appear on the bottom line of the screen, as we've noted already. There's more about this in Chapter 12.

Staying on the top row, and moving across to the right, you'll find the STOP key. This is an important key when you are running a program, or trying to make one run. Pressing this key once will make the machine stop running the program; pressing it a second time will allow the machine to continue. Unlike some computers, the MSX machines can continue just where they left off when you press this STOP key for the second time. It's a very good way of giving you time to take a close look at something on the screen, for example, without forcing you to start the program all over again to find out what happens next.

Now this is where different manufacturers have different ideas about where keys should be placed. To the right of the STOP key, you should find a group of three or four *editing keys*. As the name suggests, these are the keys that you will use when you want to correct something that has gone wrong in a program. Some manufacturers put these keys on the top line and some put them in a group on the right-hand side, near the top. They are often marked as INS, DEL and HOME. INS means INSert letters, and DEL means DELete letters. HOME means that you want to be able to type letters which will be placed starting at the top left-hand side of a blank screen. There are more details about using these keys also in Chapter 12.

Another key which is useful for correcting mistakes is the backspace key. It is marked BS by some manufacturers or labelled with a left-pointing arrow by others. If you press this key the cursor will move one space left on the screen, and will wipe out the letter that was in that place. It's a quick and easy way of correcting a mistake when you have just noticed it, before pressing the RETURN or ENTER key. On some machines you can get the cursor to move in the opposite direction, left to right, with a key that is marked TAB. On other machines this key is marked with a right-pointing arrow. This key is placed on the left-hand side of the keyboard.

Two other keys on the left-hand side are labelled ESC (or ESCAPE) and CTRL (or CONTROL). When you first start to learn about programming, you won't have much use for these keys, and I'll explain them as we go on. All of the MSX computers also use two SHIFT keys, one on each side of the bottom row of letter keys. These work in the same way as the SHIFT keys of a typewriter. When you switch on the machine ready to use, pressing a letter key gives small letters, the type which we call *lower-case*. To get capital letters, or *upper-case*, you need to press a SHIFT key along with the letter key. Either SHIFT key will do, and they are placed so that you should always have a spare finger to press one of them.

After that, it's every manufacturer for himself. Some machines have a group of keys on the right-hand side, marked with arrows. These are cursor control keys, and their effect is to move the cursor up, down, left or right, according to which key you press. Not all keyboards group these keys neatly, and you just have to find them for your own particular machine. Finally, there will be some keys on the same bottom line of the keyboard as the long spacebar key. The usual group is CAPS, GRAPH and CODE. Some machines label the CAPS key as CAPS SHIFT. It's very useful if there is a light near this key which comes on when the machine is set for capitals. Whatever it happens to be labelled, its effect is to switch capital letters on or off. If, as you type, you see lower-case (small) letters appear on the screen, then pressing CAPS or CAPS SHIFT once will make all of your typed letters appear in upper-case (capital) letters. You will still have to press the SHIFT key, however, if you want the symbols which are shown on the upper part of the number keys, and some of the keys on the right-hand side of the keyboard. Pressing the CAPS or CAPS LOCK key a second time will get you back to lower-case letters again.

The CODE and GRAPH keys are less important when you start programming. They are used like the shift key, but their effect is to provide you with a completely different set of letters and shapes. The CODE key, which can be used along with the SHIFT key, provides you with letters for other European languages, and a set of mathematical symbols, as you press the keys. Using the GRAPH key, with or without SHIFT, gives graphical symbols and some more mathematical symbols. These symbols and letters should be illustrated in the manual which comes with your computer.

Interpreting and compiling

Computing languages like BASIC can be arranged in two ways. One of these ways is called *interpreted* and the other *compiled*. The BASIC in your MSX computer is interpreted. This means that the instructions which you type are held in the memory of the computer as a set of code numbers. When you run a program, each code number for an instruction has to be interpreted. This means that the machine has to find what the code means, and then act on it. Acting on it means that the machine has to locate another set of codes in its memory, and use these to direct the microprocessor, the Z-80, which is at the heart of the computer. A language which uses interpretation is slow, because each command has to be looked up, in turn, as the computer comes to it. A compiler, by contrast, does the looking up just once after the program has been written. The instructions are then turned directly into the microprocessor codes. When a compiled program runs, then, there is no looking up to do, and the speed can be very much faster.

The language that you are provided with is interpreted BASIC, but if your MSX computer has enough memory, it is possible to insert a cartridge which will provide you with a compiled language. This language will be yet something else to learn, so you may not want to think about it at the moment. As time goes on, though, you may feel that you need to use a faster-running language. The great advantage of an MSX computer is that you have the choice.

Chapter Two
Inputs and Outputs

Chapter 1 will have broken you in to the idea that the MSX computer, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the RETURN or ENTER key is pressed. From now on, we'll use just the word RETURN to mean either of these, whatever the key happens to be labelled on the keyboard. You will also know that you can use the command LIST (then press RETURN) to print your program instructions on the screen.

There are two other useful points that you need to know before we go much further. One is that you can clear the screen by pressing the key that is marked HOME at the same time as the SHIFT key. This also has the effect of placing the cursor at the top left-hand corner of the screen. The messages at the bottom of the screen are *not* erased in this way, however. You can obtain the same screen clearing effect by pressing the CTRL key along with the L key. Pressing CTRL and L together is usually written as CTRL L to save space. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Chapter 12.

Now there are two ways in which you can use a computer. One is called *direct mode*. Direct mode means that you type a command, press RETURN, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In this mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*. The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again and then press RETURN. The set of command words that can be used, along with the rules for using them, make up what is called a *programming language*, which for your computer is MSX BASIC.

Let's now take a look at the difference between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, say 1.6 and 3.2, then you have to type:

```
print 1.6 + 3.2  (and then press RETURN)
```

You have to start with print (or PRINT) or its abbreviation ? because a computer is a dumb machine and it obeys only a few set instructions. Unless you use the word print, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like GIVE ME or WHAT IS; only a few words (about a hundred and forty of them) that we call its *reserved words* or *instruction words*. PRINT is one of these words. So that you can recognise these reserved words more easily in this book they will appear in upper-case (capital) letters from now on. You know by now, however, that you can type them in either upper-case or lower-case. You will *always* see them in upper-case when you LIST your program.

When you press the RETURN key after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8, under the command. This answer is not printed at the extreme left-hand side of the screen; it's placed one space in, and the word O.k appears under this answer. The O.k. is a *prompt* – a reminder that the computer is ready for another command. Once this command has been carried out, however, it's finished.

A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press the RETURN key. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the difference between your commands and your program instructions, however. On computers that use BASIC this is done by starting each program instructions with a number which is called a line number. This must be a positive whole number – the type of number that is called a positive integer. We looked at this idea briefly in Chapter 1.

The PRINT instruction, then, is the way that we get the computer to provide us with information. Apart from some games programs, most computer uses involve three things; inputs, processes, and outputs. The hard part about programming is not learning the language of BASIC; it's learning how to get what you want from these three actions. For the moment, we'll leave inputs aside and concentrate on outputs and a little processing.

Processing means doing whatever we want to do with numbers or words. It can be as simple as adding two numbers, or as complicated as putting a set of names and addresses into alphabetical order of surname. It can mean producing shapes and colours on the screen and moving them about. It can mean producing sound, either sound effects or music, to go with the screen displays. Summing it all up, it means all of the actions that make a computer so interesting and so useful.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used very much for calculation, but it's useful to be able to carry out calculations now and again. In addition, you are much more likely to recognise the sort of instructions that use numbers than the ones which are used to work with words. Figure

2.1 shows a four line program which will print some arithmetic results. The process here consists of four items of arithmetic, each with an output.

```
10 PRINT5.6+6.8
20 PRINT9.2-4.7
30 PRINT3.3*3.9
40 PRINT7.6/1.4
```

Fig. 2.1. A four line arithmetic program.

Take a close look at this because there's a lot to get used to in these four lines. To start with, the line numbers are 10, 20, 30, 40 rather than 1, 2, 3, 4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1,2,3, then there's no room for these second thoughts, though you can change line numbers if you have to by using the editing commands. This idea of having line numbers that go in tens is so common that it's built into your MSX computer. If you type AUTO, or press the F2 key, and then press RETURN, you will see the number 10 appear on the screen. You can then type the first line of your program. When you press the RETURN key at the end of this first line, the next line number is automatically selected for you. If you have a line of program with this number already, an asterisk appears against the line number. This automatic numbering system can save you a lot of typing!

The next thing to notice is how the number zero on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0; the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero that you see on the keyboard may be slashed or not, depending on the manufacturer of the machine, but it is on a different key from the O and is differently shaped. Type some zeros and Os on the screen so that you can see the difference.

Now to more important points. The star or asterisk symbol in line 30 is the symbol that the MSX computer uses as a multiply sign. Once again, we can't use the × that you might normally use for writing multiplication because × is a letter. There's no divide sign on the keyboard either, so the MSX computer, like all other small computers, uses /, the slash sign, in its place. This is the diagonal line which is on the same key as the question mark, *not* the one which slopes the other way and is in a different place on different MSX computers.

So far, so good. The program is entered by typing it just as you see it. You don't need to leave any space between the line number and the P of PRINT because the MSX computer will put one in for you when it displays the program on the screen. You don't need to leave a space following PRINT either, and you can type ? in place of PRINT if you wish. The MSX computer is very tolerant about this sort of thing, so don't be surprised when you see a program listing in this book which shows words run together. There are some places where you *must* leave spaces, however, and we'll deal with these as we come to them. Remember that you have only a limited number of bytes of memory available, and each space uses one of these bytes. Later on, as you become accustomed to programming, you will find that you need all the memory you can get. You'll be glad if your MSX computer already has around 28000 bytes available, or allows lots of extra memory to be added.

Getting back to the program example, you will have to press the RETURN key when you have completed each instruction line before you type the next line number. You should end up with the program looking as it does in Fig. 2.1. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. There are two things that you need to know now. One is how to check that the program is actually in the memory; the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the SHIFT/HOME keys to wipe the screen first if you like, then type LIST and press the RETURN key. When you press the RETURN key – and not until – your program will be listed on the screen. An alternative is to press the F4 key (top row) and then the RETURN key. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program operate you need another command, RUN. Type RUN, then press the RETURN key, and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.4285714285714
O.k.
```

That last line should give you some idea of how precisely the MSX computer can carry out its arithmetic. When you follow the instruction word PRINT with a piece of arithmetic like 2.8 * 4.4, then what is printed is the *result* of working out that piece of arithmetic. The program *doesn't* print 2.8 * 4.4; just the result of the action 2.8 * 4.4. You can, incidentally, obtain the whole RUN (RETURN) action just by pressing the F5 key.

This program has carried out two of the main computer actions – process and output – for you. The only input has been the program, and we can't alter any of these numbers in the program without altering the program itself. Try writing a program of this type for yourself, and see how the MSX computer carries out the calculations and displays the answers. This is all very useful, but it's not always convenient to have just a set of answers on the screen, especially if you have forgotten what the questions were. The MSX computer allows you to print anything you like on the screen, exactly as you type it, by using what is called a *string*.

```
10 PRINT"2+2=";2+2
20 PRINT"2.5*3.5=";2.5*3.5
30 PRINT"9.4-2.2=";9.4-2.2
40 PRINT"27.6/2.2=";27.6/2.2
```

Fig. 2.2. Using quote marks. In this and other examples, the abbreviation ? was used in place of the PRINT instruction word, but PRINT appears in the listing.

Figure 2.2 illustrates this principle. In each line some of the typing is enclosed between quotes (inverted commas) and some is not. Enter this short program and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2=4
```

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2=";5*1.5
```

then you'll get the daft reply, when you RUN this, of:

```
2+2= 7.5
```

The computer does as it's told and that's what you told it to do. Only a looney could believe that computers would take over the world!

This is a good point at which to take notice of something else. The line 15 that you added has been fitted into place between lines 10 and 20 - LIST if you don't believe it. No matter in what order you type the lines of your program, the computer will sort them into order of ascending line number for you. Note also that the computer has put a space between the = and the number. This doesn't always happen, though. This space is reserved for a minus sign, so if we want to make sure that a number with a minus sign is separated from the = sign, we need to add a space between the = and the final quote marks. The other important point about this example is that it shows how to make a program display what is being done. As before, the command word PRINT has to be used to make things appear on the screen, but by using quotes we can make the computer print whatever we want, not just the *results* of some arithmetic. Try making the computer print some answers for yourself, using this form of program.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as the MSX computer is concerned, always means print on to the TV screen. For activating a paper printer (*hard copy* it's called), ther's a separate instruction LPRINT (and LLIST for program listings). It's not an indication of Welsh design – the L once meant line in the days when computer printers were huge pieces of machinery that printed a whole line at a time. These instructions are of no use to you unless you have a printer connected.

```
10 PRINT"THIS IS"
20 PRINT"THE FANTASTIC"
30 PRINT "MSX COMPUTER"
```

Fig. 2.3. Using the PRINT instruction to place words on the screen, one line for each use of PRINT.

Now try the program in Fig. 2.3. Try typing the lines in any order you like to establish the point that they will be in line number order when you list the program. When you RUN the program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean print on the screen. It also means take a new line and start at the left-hand side! You will also find, incidentally, that when the words on the screen reach the line above the bottom line, all the lines apart from the bottom one appear to move up, and the top line disappears. This is the action called scrolling, and it's the way that the machine deals with displaying lots of lines on a screen which holds only 24 lines at one time, counting the bottom line.

In this example the words have been placed between quote marks, and they have appeared on the screen just as we typed them, but with no quote marks showing. This, then, is the sort of programming that is needed when you want to display instructions or other messages on the screen. The real problem, as you'll see when you try it, is getting the messages to look really neat. Nothing looks worse than printing which has split words, with half a word on one line and the rest on the next line. Even at this stage it's possible to make your printing look neat with some care. Type the PRINT" part, then type the words that you want, and continue typing *without touching the RETURN key* until you reach a position on the screen which is directly underneath the quote mark. If at this point you are in the middle of a word, then erase this word by using the BACKSPACE (BS) key. Now type a second quote mark, and then press RETURN. Start another line now, using PRINT", and then type the rest of the message in the same way. If you never cross the point where a letter comes directly under a quote mark on the line above, you will never split a word across a line end. Try it!

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using punctuation marks that we call *print modifiers*. Start this time by acquiring a new habit. Type NEW and then press the RETURN key. This clears out the old program. You might also like to use the SHIFT/HOME keys to clear the screen. If you don't use the NEW action, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line you delete any line that had that same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Fig. 2.3, for example, the line 15 that you added would be left in store even when you typed a new line 10 and a new line 20.

```
10 PRINT"This is ";
20 PRINT"the great ";
30 PRINT"MSX system."
```

Fig. 2.4. The effect of semicolons on PRINT lines.

Try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear on one line. It would have been much easier just to have one line of program that read:

10 PRINT "This is the great MSX system."

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at this sort of thing in later program examples. Meantime, look also at how I have placed a space between the last letter and the last quote mark in lines 10 and 20. The semicolon doesn't just order the computer to prevent a new line being taken; it also forces it to place one item right up against another. If you left no spaces, the words 'is' and 'the' would be printed as 'isthe', and 'great' and 'MSX' as 'greatMSX'. Try removing the spaces and see for yourself. (Turn to Chapter 12 to read how to edit out a space or any other character if you want to avoid having to type the whole line again.)

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Fig. 2.5 should not come as too much of a surprise. Line 10 contains a novelty, though, in the form of two instructions in one line. The instructions are separated by a colon (:) and you can, if you like, have several instructions following one line number in this way, taking several screen lines. So long as the number of characters in the line does not take up more than seven lines on the screen, you can put instructions together in this way. In a *multistatement* line of this type, the MSX computer will deal with

```
10 CLS:PRINT"This is MSX"
20 PRINT:PRINT
30 PRINT"-ready to work for you."
```

Fig. 2.5. Clearing the screen with the CLS instruction, and using multistatement lines. The two PRINT instructions, with nothing to be printed, cause two blank lines to appear.

the different instructions in a left to right order. The instruction CLS should not surprise you either – this clears the screen, and makes the printing start at the top left-hand corner. It's the same action as the SHIFT/HOME or CTRL L keys, but done automatically within the program. Another point about Fig. 2.5 is that line 20 causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space it's very handy. Remember that to save typing you can type ?, and the machine will convert this into PRINT for you! Now try for yourself a program which will put words on different lines like this. Remember that you have 23 lines to play with on the full screen.

```
10 PRINT1,2
20 PRINT1,2,3
30 PRINT"one","two"
40 PRINT"one","two","three"
50 PRINT"This item is too long","two"
```

Fig. 2.6. How the comma causes words to be placed into two columns.

Figure 2.6 deals with columns. Line 10 is a PRINT instruction that acts on the numbers 1 and 2. When these appear on the screen, though, they appear spaced out just as if the screen had been divided into two columns. The mark which causes this effect is the comma, and the action is completely automatic. The comma is on the key next to the letter M, so if you use the apostrophe on the key next to the semicolon key you will not get the same effect! The two can sometimes look rather alike on the keyboard, but are completely different on the screen. As line 20 shows, you can't get three columns. Anything that you try to get into a third column will actually appear in the first column of the next line down. The action works for words as well as for numbers, as lines 30 and 40 illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed *outside* the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into a column something that is too large to fit, the long phrase will spill over to the next column, and the next item to be printed will be at the start of the next line. Line 50 illustrates this – the first phrase spills over from column 1 into column 2, and the word 'two' is printed starting at column 1 on the next line.

Commas are useful when we want a simple way of creating two columns. A much more flexible method of placing words on the screen exists, however. This makes use of the command word TAB (short for *tabulate*) to position the cursor anywhere along a line, as Fig. 2.7 illustrates. For the purpose of using TAB, we need to remember that the screen is normally divided into a grid of 37 divisions across and up to 24 down (Fig. 2.8). By

```
10 CLS
20 PRINTTAB(0)"L";TAB(36)"R"
30 PRINTTAB(15)"CENTRE"
40 PRINTTAB(5)"Start here..."
```

Fig. 2.7. How TAB is used to position the cursor.

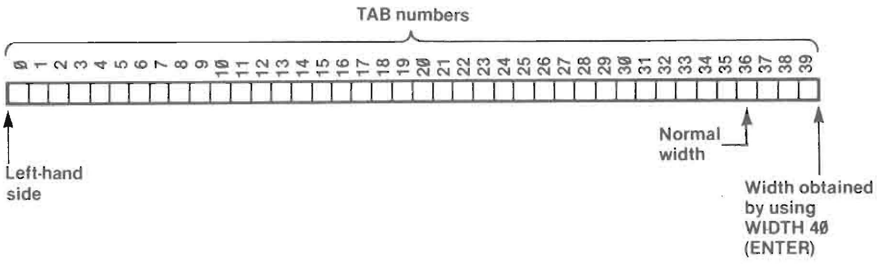


Fig. 2.8. The TAB map, showing how the TAB numbers are used. You can use WIDTH to give yourself some extra columns and KEY OFF to gain use of the bottom line.

normally, I mean when you have switched on the computer and done nothing to alter the screen arrangement. We will look later on at some commands which will alter this arrangement. These screen column and line numbers use a range of 0 to 36 across and 0 to 23 down. When you are entering programs, however, you can use only 23 lines down the screen, so that you have to keep to the numbers 0 to 22 rather than 0 to 23. The word TAB has to be followed by the number of the position, in brackets. If you omit the brackets, the TAB instruction is ignored. In Fig. 2.7 line 10 clears the screen and line 20 shows the *L*eft and *R*ight positions on the screen. L and R have been used rather than printing numbers 0 and 36 because the number 36 would have to be printed using TAB(34)! Why? Because a number is printed with a space in front, so the number 36 would take three spaces; numbers 36, 35, and 34. Line 30 prints the word CENTRE near the middle of the screen, and we'll look later at how to calculate the correct TAB number. Line 40 simply shows TAB being used to *indent* a line, which means put a space before the first word on the left-hand side.

Figure 2.9 shows some more secrets of TAB. To TAB across a line you need to use numbers that lie between 0 and 36, but you can use larger numbers! These will have the effect of getting you to a different line. It also

```
10 CLS:PRINT"Top line"
20 PRINTTAB(2)"A"
30 PRINTTAB(5)"TAB(5)";TAB(3*37+5)"Th
ree lines down"
```

Fig. 2.9. Using numbers greater than 39 for TAB. The number can be in the form of an expression.

shows that you can use an *expression* inside the TAB brackets. An expression is an item of arithmetic that has to be worked out; in the example this is 3*37+5. The 3*37 part of it means go down to the third line down, since there are 37 characters in each line. The 5 then tabs across this line to position 5, so that the phrase Three lines down starts in the same TAB position as TAB (5) above it. There are limits to this, though, because the TAB command does not allow you to use a number greater than 255. If you try to use 256 or any larger number, you'll get an error message instead. The error message is 'illegal function call', which just means that you can't do that sort of thing!

Oh, yes, how did I position the word CENTRE at the centre of a line in Fig. 2.7? This is done by calculating the correct 'across' number for the TAB instruction. The method is shown in Fig. 2.10. You have to count up

1. Count number of characters in the title, including spaces.
2. Subtract this number from 37.
3. Divide the result by 2, ignoring any remainder.
4. Use the result as the TAB number.

Fig. 2.10. The formula for centring a title, assuming the use of 37 columns.

the number of characters that you want to print centred. By characters I mean letters, digits, spaces and punctuation marks. You then subtract this from 37 and divide the result by 2. Take the whole number part of the answer – forget about any half left over – and this is the correct number to use with TAB. You can then add 37, 2*37, 3*37, or whatever you like if you need to have vertical spacing as well, provided that the number doesn't go above 255. Later, you'll see that we can use letters in place of numbers in the TAB and other instructions. This allows us to centre words without all the fuss of counting letters – but that's more advanced programming than we should be thinking about at this point! Right now, you might like to work out how you could display the words MY ADDRESS centred on the screen, with your address shown neatly printed lower down the screen. When you have achieved this you will have learned quite a lot about the use of TAB.

Figure 2.11 shows a different way of spacing out figures and letters on the screen. This uses the SPC command, and though you might think that it's

```
10 PRINTTAB(2)"MSX";TAB(16)"COMPUTING
"
20 PRINTTAB(2)"MSX";SPC(16)"COMPUTING
"
```

Fig. 2.11. Using SPC to space out your printing.

pretty much the same as TAB, its use isn't. Figure 2.11 shows how it's used. When you TAB the numbers, the first letter of each word is placed in the correct TAB position. Note, incidentally, that you can have more than one TAB in a line. I have used a semicolon to separate the second TAB, but this isn't necessary. In line 20, though, I have used SPC(16). This has the effect of putting 16 spaces between the end of MSX and the start of COMPUTING. That's quite different from having the C of COMPUTING on the number 16 TAB position. The rule is that you use TAB when you want neat columns, with the first letter of each word starting in the correct place. You use SPC if you want to fix the amount of space between words, even if the words are of different lengths. Not many computers allow you as much choice as this! Line 20 shows a semicolon between the end of MSX COMPUTING and the SPC. Once again, this isn't necessary and you can omit it. I like to use it because it makes it easier to see where the next part of the instruction is placed.

Palace of varieties

So far, our computing has been confined to printing numbers and words on the screen. That's covered two of the main aims of computing – processing and output – but we have to look now at some of the actions that take place before anything is printed. One of these is called *assignment*. Take a look at the program in Fig. 2.12. Type it in, run it, and contrast what you see on the

```
10 CLS
20 X=23
30 PRINT"2 times";X;" is";2*X
40 X=5
50 PRINT"X is now";X
60 PRINT"and twice X is";2*X
```

Fig. 2.12. Assignment in action. The letter X has been used in place of a number.

screen with what appears in the program. The first line that is printed is line 30. What appears on the screen is:

```
2 times 23 is 46
```

but the numbers 23 and 46 don't appear in line 30! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a *variable name*.

Line 20 assigns the variable name X, giving it the value of 23. Assigns means that wherever we use X, not enclosed by quotes, the computer will operate with the number 23 instead. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. Line 30 then proves that X is taken to be 23, because wherever X appears not between quotes, 23 is printed, and the 'expression' 2*X is printed as 46. We're not stuck with X as representing 23 for ever, though. Line 40 assigns X as being 5, and lines 50 and 60 prove that this change has been made.

That's why X is called a *variable* – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 2.12, providing you haven't added new lines or deleted any part of it, typing PRINT X (or PRINTX, or ?X) and pressing ENTER will show the value of X on the screen.

This very useful way of handling numbers in code form can use a *name* which must start with a letter. You can add to that letter another letter or a digit, but not spaces or punctuation marks, so that N, na, and N5 are all names that you can use for number variables, and each can be assigned to a different number. Just to make it even more useful, you can use similar names to represent words and phrases also. The difference is that you have to add a dollar sign ($) to the variable name. If N is a variable name for a number, then N$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely different.

Serenade for strings

Figure 2.13 illustrates *string variables*, meaning the use of variable names for words and phrases. It also illustrates a very different way of placing

```
10 CLS:N$="MSX"
20 A$="Computers"
30 B$="rule"
40 C$="O.K."
50 PRINTTAB(17);N$
60 LOCATE 11,5
70 PRINT"The ";N$;" ";A$
80 LOCATE7,12
90 PRINTN$;" ";A$;" ";B$;" ";C$
```

Fig. 2.13. Assigning a string variable, along with the use of LOCATE.

things on the screen without using TAB. LOCATE is a command which is used to control the cursor position. LOCATE has to be followed by up to two numbers, and these numbers have to be separated by a comma. Of these numbers, the first one is just the ordinary TAB number that you have used already. (You can, in fact, use LOCATE with just one number, and it will work like TAB. The difference is that TAB *must follow* PRINT, but LOCATE *must come before* a PRINT instruction.) The second number is a line number, and you may use numbers between 0 and 22 here. The main difference between LOCATE and TAB, apart from allowing the use of a number to select the screen line, is that LOCATE can put the cursor where you like. If you try to use TAB to move from right to left or up the screen instead of down, you are doomed to fail. LOCATE, however, will place words or numbers anywhere you like in any order. You can position words at the bottom of the screen and then work up, if you fancy that sort of thing. You can also place new words over old, which can be rather useful, as we'll see.

Lines 10 to 40 in Fig. 2.13 carry out the assignment operations, and lines 50 to 90 show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text which *must* be surrounded by quotes. You have to be careful when you mix these two, because it's easy to run words together. Note in line 70 how a space has been left between the 'e' of 'The' and the quote mark. If you omit this space, you will see the phrase 'the MSX' printed. In line 50, using TAB(17) centres MSX on the screen. Line 60 then uses LOCATE 11,5. The 11 part tabulates so that the phrase in line 70 will be centred, and the 5 selects the *sixth* line of the screen. Yes, it's the sixth line, because the top line is numbered 0, not 1. Notice that lines 70 and 90 have had to use spaces between the variable names. This has been done by typing a quote, then pressing the spacebar, then typing another quote.

You can use a string variable name for longer phrases if you like. The limit to the number of characters that you can assign to a string variable, however, is 255. Remember that you can type these string variable names as a$ and b$ and the computer will convert them to upper-case. It won't convert anything that you have typed between quotes, though. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing the phrase!

Now before you go wild on this use of variable names, a word of warning. There's nothing to stop you from using names of more than two characters if you want to. Nothing, that is, except that each letter uses up another byte of precious memory, and that it can cause confusion. The confusion can arise in two ways; one because of the 'case' of letters, the other because only two characters are significant. If you assign a variable name of N, the computer treats N and n as being identical. You can't make N=3 and n=2; if you do try this, only the assignment made most recently will stick. More seriously, if you decide to use longer names because it makes the action easier to follow, you can cause trouble. If you have NAME$="Jack", for example, you can't also have NATURE$="Sweet". The computer treats both of these as being just NA$. It doesn't bother about any letter following the second, and in this example, whichever assignment was the later one will be the one that ends up assigned to NA$.

Getting some input

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. Our programs have just consisted of a little processing and some output, but with no input apart from whatever was placed in the program. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information (number or name) into a program *while it is running*. A step of this type is called an *input* and the BASIC instruction word that is used to cause this to happen is also INPUT.

```
10 CLS
20 LOCATE5,2:PRINT"What is your name"
30 INPUT NM$
40 LOCATE 3,6
50 PRINT NM$;" -this is your life!"
```

Fig. 2.14. INPUT used to enter a name and assign it to a string variable.

Figure 2.14 illustrates INPUT with a program that prints your name. Now I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

What is your name

are printed on the screen, positioned by the LOCATE instruction in line 20. On the line below this you will see a question mark. The computer is now waiting for you to type something and then press RETURN. Until the RETURN key is pressed the program will hang up at line 30, waiting for you. If you're honest, you will type your own name and then press RETURN. You don't have to put quotes around your name; simply type it in the form that you wish to see printed. When you press RETURN your name is assigned to the variable NM$. The program can then continue, so that line 40 uses LOCATE to move the cursor to screen line 6. Line 50 then prints the famous phrase with your name at the start. You could, of course, have answered Mickey Mouse or Donald Duck or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. Even if you type nothing and just press RETURN, it will carry on with no name at all. Don't listen to the nutters who tell you that computers know everything! Now that you can type something that can be assigned to a variable, and then use the variable later, you can use all three computing actions. Could you now design a program that asks for your name, and assigns it, and then asks for your address, and assigns that? Could you then arrange it so that it then clears the screen and prints your name and address? You now know all of the commands that are needed.

We aren't confined to using string variables along with INPUT. Figure 2.15 illustrates an INPUT step which uses number variables A and B. The same procedure is used. When the program hangs up with the question mark appearing, you can type a number and then press the RETURN key. This time, though, the question mark will appear on the same line as the message.

```
10 PRINT"Enter a number ";
20 INPUT A
30 PRINT"..and another number, please
";
40 INPUT B
50 PRINT:PRINT"Product is ";A*B
```

Fig. 2.15. Assigning numbers with INPUT.

This is because there are semicolons following the PRINT messages in lines 10 and 30. The action of pressing RETURN will assign your number to A and allow the program to continue. Lines 30 and 40 then get another number from you, and line 50 proves that the program is dealing with the numbers that you entered. This is a simple example of the computer used with input, process and output. The input steps obtain your numbers and assign them, the process is just multiplication, and the output is the action of printing the product. You could design a program for yourself which adds or subtracts or divides numbers to order in this way. When you use a number variable in an INPUT step, what you have typed when you press RETURN must be a number. If you attempt to enter a string, the computer will refuse to accept it. Some computers stop running at this point, but the MSX computers simply print 'Redo from start', and this gives you another chance to type a number and press RETURN again. If your INPUT step uses a string variable, then *anything* that you type will be accepted when you press RETURN.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 2.16 shows an example – but with INPUT used in a different way. This time,

```
10 CLS
20 INPUT"Type your name, please ";NM$
30 PRINT
40 PRINT"Thank you, ";NM$;" _"
50 PRINT"I'm very pleased to meet you
."
```

Fig. 2.16. Using INPUT to print a phrase as well as accepting a string.

there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name NM$. This line 20 has the same effect as the two lines:

```
15 PRINT "Type your name, please ";
20 INPUT NM$
```

Again the question mark appears on the same line as the question, and your reply is also on the same line – unless the length of your name causes letters to spill over on to the next line.

The use of INPUT isn't confined to a single name or number, however. INPUT can be used with two or more variables, and we can mix variable types in one INPUT line. Figure 2.17, for example, shows two variables being used after one INPUT. One of the variables is a string variable NM$;

```
10 CLS
20 INPUT"Name and number, please ";NM
$,N
30 PRINT:PRINT
40 PRINT"The name is ";NM$
50 PRINT"The number is ";N
```

Fig. 2.17. Using more than one variable with INPUT.

the other is the number variable N. Now when the computer comes to line 20, it will print the message and then wait for you to enter both of these quantities; a name and then a number. There are two ways of entering these quantities. One way is to type the name, then a comma, and then the number. Pressing the RETURN key will then assign the two variables in one operation, and the computer will continue on its way. The other method consists of entering each quantity separately. If you type the name and then press RETURN the computer will print *two* question marks on the next line. This is a symbol meaning 'more needed', and that's a signal for you to type the number and then press RETURN again. Whichever way you use, the name and number will be printed again in lines 40 and 50.

Now suppose that you wanted to write a program that would total sets of four numbers for you. You could have an INPUT step which used four variable names. Remember that you can have names like N1, N2, N3, and N4 if you like. You could write such a program now. Remember to clear the screen, and print some sort of message to remind you what to do. You'll need an INPUT to get the numbers assigned to variables, and then a PRINT step to say what is being printed and then print the sum of the numbers. Can you make this look neat? One more point. When you use INPUT, you can't type anything which contains a comma because the machine will assume that this means another variable is being used. If you want an INPUT step to accept *anything* that you type, use LINE INPUT in place of INPUT.

Reading the data

There's yet another way of entering data into a program while it is running. This involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items on the list can be separated by commas. Each time an item is read from such a list a *pointer* is altered, so that the next time an item is needed, it will be the next item on the list rather than the one that was read the last time round.

We'll look at this in more detail in the form of examples later, but for the moment we can introduce ourselves to the READ...DATA instructions. Figure 2.18 uses these instructions in a very simple way. Line 20 reads the

```
10 CLS
20 READ NM$
30 PRINTNM$;
40 PRINT" is valued at ";
50 READ N
60 PRINTN
70 DATA Disk drive,190
```

Fig. 2.18. Using READ...DATA to take items in order from a list. There should be a pound sign immediately before the last quote mark in line 40.

first item on the list and assigns it to the variable NM$. This is printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the number which is the second item in the list. This is assigned to the variable name N (we could just as easily have used NM$) and printed in line 60. This is a good example of why it is so useful to be able to keep printing in one line with semicolons. You can print each bit as it is read, and still end up with a complete line. Note that you *don't* need any quotes around the name in the DATA line 70. You do, however, have to be careful about how you match your READ and your DATA. If you use a number variable in the READ, like READ A, then what is in the DATA line being read *must* be a number. If you use a string variable, as in READ A$, then it doesn't matter whether your DATA line contains a number or a string. Remember, though, that if you read a number using READ A$, then you can't carry out any arithmetic on that number. Later on, we'll see how a number in this form can be converted back to number form so that you *can* carry out arithmetic.

The READ...DATA instructions really come into their own when you have a long list of items that are read by repeating a READ step. These would be items that you would need every time the program was used, rather than the items you would type in as replies. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now.

The USING modifiers

The PRINT instruction can be made even more useful by adding the word USING. This is a more advanced topic, and if you are reading this book for the first time, it's just as well to skip the rest of this chapter. You can come back to it later when you have more experience with placing text and numbers on the screen.

The thinking behind USING is that you often want numbers or strings to be arranged in a particular way. We call this *formatting*, so USING is a formatting instruction. Suppose, for example, that you were writing a program which had to calculate the amount of VAT on a price. This means finding 15 percent of the price and adding on this amount of tax. It's simple enough, but what do you do when the result looks silly? For example, the VAT at 15 percent on a price of £2.13 is £0.3195. Now you can't add these to get £2.4495 and print this as a price. Formatting offers one of several ways to get rid of this problem. We might, for example, want to print the final price to two decimal places only, i.e. as £2.45. This is where print formatting comes into its own. We can use symbols in place of the digits of a number to show the computer how we want the number printed. These symbols are placed into a string, and this string *or string name* is used following PRINT USING.

Figure 2.19 shows how this can be used. Line 10 obtains a quantity from you, and line 20 calculates 15 percent (which is .15) of it. Line 30 assigns C$

```
10 INPUT"Price, please ";A
20 B=.15*A
30 C$="##.##"
40 PRINT""; USING C$;A+B
```

Fig. 2.19. An illustration of PRINT USING, showing how amounts can be rounded. There should be a pound sign between the quotes in line 40.

to "##.##". This means that a number is to be printed with at least two digits before the decimal point, and only two following. Line 40 then prints the result. You'll see that if you enter 2.13 as a price, you get the result £2.45. If you enter a price which uses more than two places of figures *before the point*, then the result is printed with a % sign in front. This is just a reminder that the number is greater than you allowed for. Try altering line 30 to read C$="####.##", and then see how the result of entering 2.13 is printed. The spacing is there because you have requested it in the format.

There are many formatting marks that can be used in this way, and your manual will probably list them all. At the moment, the only other important one is the one that places a pound sign in the correct place. If you use a formatting string such as ££###.##, then the effect is to place *one* pound sign in front of the number. In this case, no spaces are left, because this is the formatting for cheque printing, and you wouldn't want to leave a space between the pound sign and the rest of the numbers. Figure 2.20 shows this in action. Line 10 shows six hashmarks in the formatting string, but you'll see that if you try small amounts in the INPUT line, each sum is printed with no space between the pound sign and the first number. Now you can start to design the output stage of that accounts program!

```
10 C$="######.##"
20 INPUT"Sum to be paid, please ";S
30 X=S+.15*S
40 PRINT:PRINT
50 PRINT"Please pay ";USING C$;X
```

Fig. 2.20. Putting a pound sign automatically into money amounts. There should be two pound signs at the start of the string of hash marks.

Chapter Three
Quantities of Numbers

So far, we have looked at how a number can be printed and how it can be assigned to a variable name. Many uses for computers operate with numbers (yes, even games programs), and so we have to know something about how the MSX computer deals with numbers. The more specialised instructions that are needed by engineering and scientific programs, however, have been left to the end of this chapter so that you can take them or leave them. In general, unless your programming is likely to be concerned with engineering or scientific problems, you can leave them.

We can use numbers in programs in two different ways. One is the use of numbers as *constants*. A constant is a number whose value can be put into a program at the programming stage and never changed. You might, for example, be writing a program that deals with VAT at 15 per cent, so that the number .15 keeps appearing. The other way of handling numbers is to assign them to variable names. For most purposes, this is a better way of dealing with them than the use of constants. There are two reasons for this. The first is that a variable can be changed easily. Suppose you have a program in which the amount .15 occurs a dozen times. It's a lot easier to have a line which reads T=.15 early in the program, and which uses T rather than .15 each time a calculation is carried out. That way you only have to type T, not .15. Also if the rate of tax changes (usually meaning *increases*), then you only have to alter the line that reads T=.15, not all the places in the program where T is used. That's a very great advantage. The other point is one we're coming on to; that a variable can often need less memory for storage than a constant.

MSX machines, unlike most computers, allow you to use three different types of number variables. At first, when you are fairly new to computing, the differences do not seem to be very important. Later on, though, you will find that if you are aware of these differences, you can take advantage of them to make your programs run more efficiently. The differences concern the amount of memory that is needed to store each value, the time that the computer takes to work with each value, and the precision of working with each value. The three types of variables are integer, real and double-precision.

Integer variables

An integer variable is one which uses a name starting with a letter, as usual, and can have a second letter or digit. It ends with the percent sign %, so that names like N%, X2%, TT% are all valid integer names. Unlike other number variables, though, integer variables obey very strict rules. The value that is assigned to an integer variable must not contain any fractional part. You can correctly assign A%=5, for example, but not A%=5.5. If you *do* use A%=5.5, then you will find from using PRINT A% that the value which has actually been assigned is 5! The fraction has been completely ignored. In addition, integer variables can use only a limited range of values, which can range from −32768 to +32767. If, for example, you try to assign A%=−32800 or A%=42000 you will get the message 'Overflow' when the program tries to carry out this command. Overflow means that there isn't enough memory to take the number, because an integer number uses very little memory – less than any other kind of variable. An integer variable value needs only two bytes of memory for storage, though each integer will, in fact, use up five bytes of memory. This is because one byte is kept to place a code for the variable type, and two more are used for the two characters of the variable name, making a total of five.

Because so little memory is used, any program which can make use of only integer variables will run faster and take up less memory than a program which uses the other variable types. If you can be sure that your programs will use only integer number variables, there are two ways that you can take advantage of this. One is to remember to place the integer mark, %, after each number variable name. The other, which is easier when you are using a large program, is to define all your integers *in advance*. You can do this by using DEFINT. For example, if your program starts with:

```
10 DEFINT J-N
```

then any variable name which starts with J, or with K, L, M, or N, will be an integer. This includes names like N2, JJ, LQ, Kim, and so on. It's the *first* letter that counts here. Very few computers have this useful DEFINT command nowadays, and only old-timers who used the TRS-80 will remember it. You can override the action of DEFINT if you like by using one of the other symbols. For example, even if you have used DEFINT J–N, you can still assign a number like K1!=101.76. In this case, K1! is a *single-precision* or *real* number, not an integer.

Integers can be used with any arithmetic action, but it's better to stick to addition, subtraction and multiplication. The reason is that division, along with actions like taking square roots, can give results which can contain fractions. Integer numbers cannot make use of fractions, so something has to give. Take a look at Fig. 3.1 to illustrate this. The first four lines assign integer values to integer variable names. Because the first line has used DEFINT A–Z, all number variables are integer variables. Line 40 has

```
10 DEFINT A-Z
20 P=45
30 Q=12
40 D=P/Q
50 PRINTD
60 PRINTP/Q
```

Fig. 3.1. Using DEFINT to declare integers and some integer actions.

assigned D as the result of dividing P by Q. Now this result is *not* an integer, so if it is assigned to an integer variable, the fractional part will be ignored, as line 50 shows when the program runs. The PRINT action, however, is *not the same as assignment*, and so the correct value is printed. If you use division in a program that makes use of integer variables, then, you can PRINT the result but you can't be sure that it will be correct if you assign it to an integer variable. It's surprising how many numbers that you deal with are integers. The most obvious use is in a count, but integers can be used for other calculations. If you have a program which deals with small sums of money, for example, you can convert each amount to pennies and then use an integer variable. The program will not be able to handle a sum of more than 32767 pence, which is £327.67, but it can sometimes be quite useful.

Real number variables

A real, or single-precision, number variable can be marked in two different ways. One way is to make use of the exclamation mark (!) following each variable name. (Names like A!, BC!, H2! are all valid real number variable names.) The alternative is to define a range of variable names as single-precision. This is done by using DEFSNG. For example, DEFSNG A-Z will define all number variables as single-precision, unless some other mark is used following the variable name. A variable like this needs a total of seven bytes to store, of which only four bytes are used for the value. This allows only six figures of the number to be stored, so if you assign a number which uses more than 6 figures it will be approximated. For example, if you assign A!=12345678, this value will be stored as 123457 and 100, meaning that 1234567 will be multiplied by 100 to give the (approximately) correct value. If you carry out PRINT A!, then the answer will appear as 12345700. Numbers of this kind, then, are accurate only for values up to 999999 and down to −999999. You can still get accurate results with numbers like 134000000 because the zeros are not 'significant'; they are not the part of the number that is coded into six digits. Similarly, numbers like .00000012 are stored accurately, because once again, the zeros are not part of the number that has to be stored in coded form. Many computers use a different form of coding which allows a larger number of figures, but is not so accurate for any of the numbers. The MSX system ensures that arithmetic which uses single-precision numbers will be carried out with complete accuracy if the numbers contain six significant figures or less. If you are interested in number systems, then the technical reason is that MSX computers store numbers in BCD form rather than in binary fraction form.

Double-precision

A variable name which is unmarked, or which ends with the hash mark (#), or which has been selected by the use of DEFDBL, is a double-precision variable. A variable of this type needs a total of eleven bytes for storage, of which eight bytes are used to contain the number value. This allows numbers to be stored precisely up to a maximum of fourteen significant figures. Even if you do not use the # mark or DEFDBL, *any number variable will be stored in double-precision form* unless you mark it as an integer or as a single-precision variable. This is what is called the *default* case, meaning that you don't have to do anything special to have your variables stored in double-precision. This is why MSX machines produce so much more accurate results in arithmetic than so many other computers.

The trouble with double-precision, though, is that it takes quite a lot of memory, and a lot of work for the computer to work with the numbers. Throughout this book, then, we'll use only as much precision as we need, so you will see the use of the % and ! marks in many of the programs. You should really use double-precision only where it is important - for accountancy, where answers *must* be exact, and for a few problems in science and engineering where high precision is essential. Oddly enough, such high precision is seldom really needed.

Handling numbers

The amount of computing that we shall cover in this book will persuade you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for word processing or even for accounts. It's time, then, to take a look at the number abilities of MSX computers. It is a comparatively brief look because we simply don't have space to explain what all the mathematical operations do. In general, if you understand what a mathematical term like sin or tan or exp means, then you will have no problem using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of *incrementing* if you are counting up and *decrementing* if you are counting down. Incrementing a number means adding one to it; decrementing means subtracting one from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.2 shows.

```
10 CLS
20 X=5
30 PRINT"Value of X is ";X
40 X=X+1:PRINT
50 PRINT"Now we've used X=X+1":PRINT
60 PRINT"The value of X is ";X
```

Fig. 3.2. Incrementing, using the equals sign to mean becomes.

Line 20 sets the value of variable X as 5. This is printed in line 30, but then line 40 increments X. This is done using the odd-looking instruction X = X + 1, meaning that the new value that is assigned to X is one more than its previous value. The rest of the program proves that this action of incrementing the value of X has been carried out.

The use of the = sign to mean becomes is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

X = X−1

and this would have the effect of making the new value of X one less than the old value; X has been *decremented* this time. We could also use X = 2*X to produce a new value of X equal to double the old value, or X = X/3 to produce a new value of X equal to the old value divided by three. Figure 3.3 shows another assignment of this type, in which both a multiplication and an addition are used to change the value of X.

```
10 CLS
20 X=5:PRINT"X is now ";X
30 PRINT
40 X=2*X+4
50 PRINT"It's changed-"
60 PRINT"X is now ";X
```

Fig. 3.3. A more elaborate reassignment using an expression.

Number operations

An operation means some action that is carried out on the value of a variable. For example, if variable X has been assigned with a value, then 2*X is an operation; the operation of multiplying the value of X by two. Similarly, X−1, X/2 and X+3 are all operations which are carried out on the value of X. Another important operation often carried out on numbers is raising to a power (also called *exponentiation*). The ^ symbol is used by MSX machines for this purpose, so that 5^2 means five to the power of two, or five squared, and its value is 25. To take another example, 3^3 is three to the power of three, or three cubed, whose value is 27. You can use fractional or negative powers in this way. The meaning of a fractional power is a root, so that 4^(1/2) means the square root of four, and 27^(1/3) means the cube root of 27. We make use of square roots so often that a special operator SQR can be used in place of ^.5, so that you can type SQR(4) in place of 4^.5 or 4^(1/2). The result, whether you use PRINT SQR(N) or X=SQR(N) will be a double-precision number. A negative power means the *inverse* of the power of the number, so that 5^−2 means one over five squared, which is 1/25, and 3^−3 means one over three cubed, which is 1/27. A less obvious operation is X=−X. This means that the value of X is to be made negative. If X had the value of +5, then the action of X=−X would make X equal to −5; if X had the value of −5, then X=−X would make the value +5. This operation is called *negation*, or *unary minus*.

These operations can be carried out on numbers, or on variables which have been assigned to number values. Number operations cannot be carried out on string variables, however. Figure 3.4 lists the operations which are of

Order of priority

For ordinary arithmetic, order of priority is MDAS - multiplication and division, followed by addition and subtraction. The full order of priority is:

1. Raising to a power, using ^
2. Multiplication and division
3. Addition and subtraction
4. Comparison, using = <>
5. AND
6. OR
7. NOT

Fig. 3.4. The simpler number operations in order of precedence.

importance to you at this stage. These operations are given in order of precedence, and this is the next subject that we have to devote some attention to before leaving the subject of number operations.

When we deal with one operation at a time, programmed in separate lines, the order in which operations are carried out is strictly the order of the line numbers. For example, if you program:

```
10 X=12.5
20Y=2.4*X
30Z=Y-4
```

then the order of actions is assignment, then multiplication, and then subtraction. It may not be so easy to see what would happen if you programmed Z=2.4*X−4. As it happens, this gives the same answer, because X is multiplied by 2.4 and then 4 is subtracted. You could imagine, however, that the command could be interpreted as subtracting 4 from X and then multiplying the result by 2.4. Take another example – what would you expect to be the result of:

5.4−2.2*3+4.6/1.5−2.2*1.2

If your reply is −.7733333333333, then you have taken the numbers in the correct groupings. You do *not* carry out the operations in the order in which they are written. Instead, you carry them out in the order of precedence, which is multiplication, division, addition and subtraction; MDAS if you need a short way of remembering it. What this means is that multiplication and division steps are *always* carried out first, followed by the addition and subtraction steps. The computer will always use this order of precedence unless you override it by the use of brackets. Figure 3.5 shows the full order of precedence including unary minus and exponentiation operations. Note that the equality = has quite a low order of precedence.

---

1. Anything that is enclosed in brackets. Innermost brackets have highest precedence.
2. Functions, such as SQR, SIN, LOG.
3. Raising to a power, using ^.
4. Negation – use of the negative sign.
5. Multiplication and division.
6. Integer division using \.
7. Remainder using MOD.
8. Addition and subtraction.
9. Comparison, using = <>.
10. NOT.
11. AND.
12. OR.
13. XOR.
14. EQV.
15. IMP.

Note: XOR is seldom used, and EQV and IMP even less.

Fig. 3.5. The full order of precedence for all number operations.

There are often times when you want to impose an order of precedence which is not the same as the machine's. For example, you may want to subtract 4 from the value of X and then multiply by 2. If you programmed this as 2*X−4 you would not achieve this, because the order of precedence would ensure that the value of X was multiplied by 2 *before* the number 4 was subtracted. Similarly, if you typed X–4*2, the computer would multiply 4 by 2, giving 8, before subtracting this value from the value of X. If you want to make the computer carry out a subtraction before a multiplication you have to enclose the subtraction in brackets. By typing (X–4)*2, you *force* the operation of X–4 to be carried out first and the multiplication second. This is because the use of brackets has a higher order of precedence than multiplication. Whatever you have placed within the brackets will be carried out in normal order of precedence. For example, if you have:

```
(5–1.2*X)–(4+1.6/Y)
```

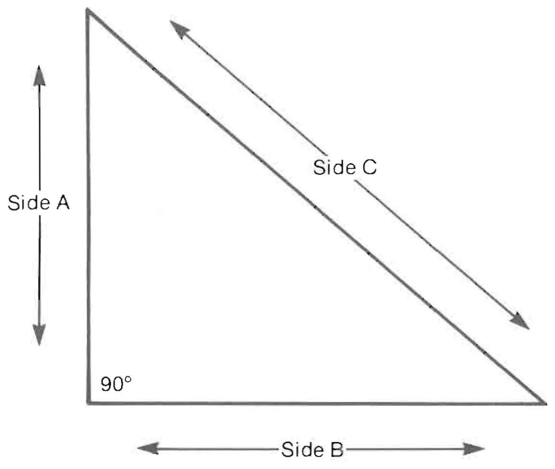
the order of actions will be to work out the contents of the brackets first, in normal precedence. The value of X will be multiplied by 1.2, and this result will be subtracted from 5. The value 1.6 will be divided by the value of Y, and 4 will be added. The value of the result of the second bracket will then be subtracted from the value of the result of the first bracket to obtain the final result. When one set of brackets is contained inside another (nested brackets), then whatever is enclosed in the innermost brackets will be carried out first. When this has been done the action of the outer brackets will be executed, and following that, anything that was placed outside all the brackets. This order of precedence of brackets applies to *all* operations, not just number operations, as we shall see illustrated later when we deal with strings.

Formula translation

The first uses of computers were for solving equations, and one of the first useful computing languages was FORTRAN, an abbreviation of FORmula TRANslation. FORTRAN is still used to some extent in mainframe computers, but BASIC is just as capable of working with formulae, and many other computing languages have been devised with similar capabilities. If all you wanted to do was to work out one single answer from a formula, then a calculator would be the quickest and most obvious method. Where the computer becomes essential is when the formula is a large and tedious one, and a very large number of answers have to be worked out.

Take a simple example first, Fig. 3.6. The quantities A and B represent the lengths of the short sides of a right-angled triangle (remember Pythagoras?) and C represents the length of the long side. These sizes are strictly related by the formula which states that C squared equals A squared plus B squared. The value of C is thus found by taking the square root of each side of the equation, so that C is equal to the square root of A squared added to B squared. Suppose that we have assigned number values to variables A and B, how do we type a line which will give the value of C?

The important thing to remember is precedence. The square root action



Whatever the size of the triangle:

$$C^2 = A^2 + B^2$$
so that $C = \sqrt{A^2 + B^2}$

This is programmed as C = SQR(A^2+B^2)

Fig. 3.6. The right-angled triangle formula and how to program it.

will *always* take precedence over addition, so that if we program this as:

C=SQR(A^2)+B^2

the value of A will be squared and then rooted (leaving it unchanged) and added to the value of B squared. This is not what we want. What we have to do is carry out the addition of the squares completely within brackets, and have the SQR preceding the brackets:

C=SQR(A^2+B^2)

The squaring is thus carried out first, then the addition, and finally the square root action.

Formula translation will very often include the use of trigonometrical quantities such as the SIN, COS and TAN of angles. It is important to remember that these trigonometrical actions will have very high precedence, so that a number value for a SIN, COS or TAN will be found even before anything else inside a bracket is worked out. SIN, COS, TAN, and all of the others are called *functions* of angles, meaning that they are a method of finding a value for a number. The computer uses many of these functions, each of which is said to 'return a number'. By this, we mean that the result of the action will always be a number if the function has been used correctly. MSX computers, in common with others, assume that the values of angles will be in units of radians, and Fig. 3.7 shows the relationship between radians and the more commonly used degrees for angular measure. MSX machines have no built-in converter for obtaining radian values from degrees, or degrees from radians.

The exact equivalence between degrees and radians is that 180 degrees are equal to PI radians. If we take PI as approximately 3.14, then one radian is about 57.3 degrees, and one degree is about 0.017 radians.

For program use, it's best to define a variable PI!=3.14159. You can then use several common angle values in radians as follows:

Degrees	Radians
30	PI!/6
45	PI!/4
60	PI!/3
90	PI!/2
180	PI!

Fig. 3.7. Converting between degrees and radians.

Number functions

Figure 3.8 illustrates the use of some of these number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 10 picks the value of 2.5 for X. Line 20 then

```
10 CLS:X=2.5
20 PRINT"X squared is ";X^2
30 PRINT
40 PRINT"Its square root is ";SQR(X)
50 PRINT
60 PRINT"Its natural log. is ";LOG(X)
70 PRINT"..and its ordinary log. is "
;LOG(X)/2.303
```

Fig. 3.8. Using some number functions.

prints the value of X squared, meaning X multiplied by X. This is programmed by typing X^2; the character which MSX computers use for this is on the 6 key. To obtain the square root of the number that has been assigned to X we use the instruction word SQR. An alternative is X^.5, but SQR(X) is easier to type and remember. For other roots, like the cube root, you can use expressions like X^(1/3) and so on. LOG(X) produces the natural logarithm of X. This is not the type of logarithm that you may want, and to find the ordinary (base 10) log you have to divide this result by 2.303. This needs some other alterations if you are working with the logarithms of fractions, but you'll have to consult a mathematics textbook for that one; there isn't room here.

Figure 3.9 illustrates the various number functions that can be used, with

ABS (X)	Converts negative sign to positive
ATN (X)	Gives angle (in radians) whose tangent is X
CINT(X)	Converts X to an integer
CDBL (X)	Converts X to a double-precision number
COS (X)	Gives the cosine of angle X (radians)
CSNG (X)	Converts x to a single-precision number
EXP (X)	Gives the value of e to the power X
FIX (X)	Strips fraction from X
FRE (X)	Gives amount of memory not in use or reserved
INT (X)	Gives the whole-number part of X
LOG (X)	Gives the natural logarithm of X
RND (X)	Gives a random fraction between 0 and 1
SGN (X)	Gives the sign of X. The result is +1 if X is positive, −1 if X is negative, 0 if X is zero
SIN (X)	Gives the sine of angle X (radians)
SQR (X)	Gives the square root of X
TAN (X)	Gives the value of the tangent of angle X (radians)

Fig. 3.9. Number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

a brief explanation of what each one does. Some of these actions will be of use only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs. We'll look at a few of them now, though, because they will be used in many of the program examples that follow. RND is a function that is particularly important in many games and graphics programs. RND means random, and if you assign X=RND(1), then what you get is a number which has been picked at random, and which is something between 0 and 1. It will never be quite zero, or quite 1, however. The main thing when RND is used, is to ensure that it does not produce the same sequence of numbers each time the program runs. This is because the number is calculated from a starting number which is called a *seed*. Normally the seed will be the same each time a program runs. To get round this, make one of the early instructions in a program X=RND(−TIME), and this will have the effect of setting a different seed each time you use the program.

Another function that comes in useful here is INT. INT causes only the integer part (the whole-number part) of a number to be used. It's a way of chopping off fractions, and it's very often used along with RND. For example, suppose you want a random number which lies between 1 and 10. The method of obtaining this is:

```
X%=INT(RND(1)*10+1)
```

The RND part will generate something greater than 0 and less than 1. Multiplying by 10 will give something which can take values from less than 1 to less than 10. Adding 1 gives a range of just over 1 to less than 11. Taking INT then gives a range of 1 to 10.

SGN is a function that can return −1,0, or +1. It is used in the form X%=SGN(Y). If Y is negative, then X% will be −1. If Y is positive, X% will be +1. If Y is zero, then X% is also zero. It's a good way of finding out what the sign of a number is. You might, for example, want a routine that rejects negative numbers, perhaps because you want to take square roots.

Defined functions

The *predefined* or *intrinsic* functions such as SIN, COS and TAN, which are programmed into the MSX machines, are often all that will be needed to solve an equation. A few machines, including all MSX machines, allow an extension of the use of functions, which is the creation of a new function from existing ones. This is a particularly useful way of programming a formula, and it is known as the *user-defined* function.

A user-defined function is particularly helpful in a program which makes much use of an action. Suppose, for example, that we frequently need to find the root of the sum of squares in a program. Instead of programming C=SQR(A^2+B^2) each time, we can define a function of our own, which will be called RT. If we want to find the root of the sum of A^2 and B^2, then all we need to do is use the function RT. This is done in the form FN RT(A,B). The FN part is used to indicate to the computer that a defined function is to be used. RT indicates which (of possibly more than one) function is to be used, and A and B are the numbers (or *parameters*) which the function will need to use. The important thing about a defined function as used in the MSX machines, is that you are not confined to the use of variable names A and B. Your variables could be X and Y, or P and Q; any pair of variable names that you like. The only condition is that both variables must have been assigned with number values before the defined function is called into use. By making use of a defined function, then, a formula need be typed once only, and then subsequently used in a program by making use of a function name such as RT. You must be sure that the name which you pick is not the name of an existing function such as SIN, COS, TAN etc.

Figure 3.10 shows a very simple example of a defined function in use. Line

```
10 CLS
20 DEF FNRT(N)=SQR(N)
30 INPUT "Number, please ";A
40 PRINT"Its square root is ";FNRT(A
)
50 GOTO 30
```

Fig. 3.10. Using a defined function, FNRT.

20 starts with DEF to indicate to the computer that this is the definition of the function. This is followed by the name of the function, FNRT, and the parameter name which will be used, N in this example. This definition of what FNRT is *must* be made before the function is used. Some machines allow such a definition to be put at the end of a program, but for MSX machines you *must* place it near the start. Line 30 then asks you for a number, and line 40 prints the value of the square root of this number. This is because FNRT was defined as SQR(N), the square root of a number. The important point here is that the definition in line 20 uses a variable N, but when we use the function the variable name is A. This is a very important point about defined functions, and one which makes them very useful. If the function were able to make use of only the variable name N, then any other variable value that we wanted to use would have to be reassigned. For example, if our value were held as variable X, we would need a line such as:

1000 N=X

before we could find the root value. Using the form of defined function that MSX computers permit avoids this extra action, which is called passing variable values. The action of finding roots repeats, because of the GOTO 30 command in line 50. This ensures that the lines 30 to 50 are repeated until you press the CTRL and STOP keys together.

Now look at another slightly more complicated defined function in Fig. 3.11. This time, the defined function works out the square root of the sum of

```
10 CLS
20 DEF FNHYPOT(A,B)=SQR(A^2+B^2)
30 INPUT "Two numbers, please ";X,Y
40 PRINT"Third side is ";FNHYPOT(X,Y)
50 GOTO 30
```

Fig. 3.11. Another defined function, this time using two quantities (parameters).

two squares, using two numbers which in line 20 are referred to as variable names A and B. Line 20 also gives the definition of the function as equal to SQR(A^2+B^2), the familiar root of the sum of the squares. The definition of the function *must* be completed in one line. In line 30, two numbers are asked for. When you type the two numbers and use RETURN, the next line will print the value of the root of the sum of the squares. Note once again that the numbers are represented by variable names X and Y, and we can 'call' the function by using FNHYPOT(X,Y), even though the definition used variable names A and B. The use of a defined function is a very useful way of making the computer carry out the actions of applying a formula to a set of numbers. The numbers are assigned to variable names, and then put into the defined function formula by using an instruction such as FNformula(A,B,C,D), using as many variables as you need numbers to put into the formula. The only thing you have to watch is that you use the correct number of variables. If your DEF FNformula uses four variables, then your FNformula must also use four – no more, no less.

Tests and approximations

When you make use of number functions in formulae, or to create defined functions, you always have to be careful about approximations. If you don't mark number variables with a ! or % sign, then you will be working with double-precision numbers, and the accuracy will be very good indeed, certainly good enough for all but the most exacting requirements. What you have to watch, though, is that the formula is correct and that you have programmed it correctly. If you have spelt an instruction word wrongly or used it in the wrong way, you will get a 'Syntax error' message when the program runs. If you have made a mistake in programming the formula, but have used the instruction words correctly, there will be *no error messages.* The answers, however, will be wrong! You won't know this unless you test the program *using numbers that will produce an answer that you know*. For example, if you are testing a defined function for the root of the sum of squares, you would test with numbers like 3 and 4, or 5 and 12. The result from 3 and 4 should be 5, and if you get 4.999999999999, something is not correct. The MSX computer should not approximate numbers of this size. Similarly, if the result is a long way out, there has to be something wrong in the programming. Testing like this is very important if the formula is a complicated one, because if you can't rely on the computer to give correct results there isn't much point in using a computer!

Number arrays

So far, we've looked at variable names for numbers with the idea that we might be using just a few numbers in our programs. Suppose that we want to use *hundreds* of numbers, though? Just to give an example, suppose we are keeping records of the amounts of money paid into a Christmas Club fund. There might be a couple of hundred members of the Club, each one contributing a different amount. How do you keep records of this? You can't very well use a different variable name, like A, B, C ... for each member. This would use up too many letters, and you would find it difficult to keep track of the variable names. An application like this requires a rather different use of a number variable.

This new use is a *number array*. A number array just means a set of numbers that you want to keep together as a set. Examples such as Christmas Club payments, subscriptions, examination marks, scores in league matches, are all numbers that belong to a set. The computer allows you to use just *one* variable name for a set like this. You then distinguish between one item and the next by using a 'reference number', which is called a *subscript*. The subscript number follows the variable name, within brackets. For example, you might want to use the variable name of Sb for subscriptions to a Club. The first member's subscription is assigned to variable Sb(1) (called ess-bee of one), the next to Sb(2) (ess-bee of two) and so on. The important point is that the number – 1, 2, or whatever – can be a number variable. We can refer to Sb(X), and make X equal whatever we like. If you want to find what member number 27 paid this year, just make X=27 and PRINT Sb(X), and it's on your screen!

This very different way of handling numbers in lists is something that we'll come back to. The reason is that arrays are best handled by a program that allows actions to be repeated, and we're not at this stage yet. Watch out for more in Chapter 5!

Chapter Four
Strings Attached

In Chapter 3, we took a look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them so is that the really eyecatching and fascinating actions that the computer can carry out are so often done using string functions. A string means any collection of up to 255 characters – the sort of thing that we put between quotes in a PRINT action, or assign to a string variable name like A$, or BC$. What's a *string function*, then? As far as we are concerned, a string function is any action that can be carried out with strings. That definition doesn't exactly help you, I know, so let's look at an example, in Fig. 4.1.

```
10 CLS
20 A$="ONE"
30 B$="TWO"
40 PRINTA$+B$
50 A$="12":B$="34"
60 C$=A$+B$
70 PRINTC$
```

Fig. 4.1. Assigning and concatenating (joining) strings. This is not the same action as addition of numbers.

This shows two strings, A$ and B$, being assigned in lines 20 and 30. A$ is assigned to "ONE" and B$ to "TWO" – remember that you must use quotes in an assignment like this. Line 40 shows what you get for A$+B$. What is printed on the screen is ONETWO; the two strings run together. The + sign, then, is a kind of operator for strings, but the operation is not addition in the way that we add numbers. To distinguish it, this use is called *concatenation*. The rest of the program shows that concatenation works in the same way even if the strings are of number quantities. If you PRINT A$ or PRINT B$ after line 50 has been run, you will see 12 for A$ and 34 for B$, but C$ is 1234, not 46. The + is not an addition sign as far as strings are concerned; it is a joining sign. Concatenation can be useful if you have carried out actions on two different strings and you then want to join them. Suppose, for example, that you have a mailing list program, and to save on memory space you allow names of up to ten characters only. When a name is entered, you don't chop off all the characters after the tenth. This would result in JONATHAN MILKMAN being chopped to JONATHAN M because the space counts as a character. The more sensible method is to separate the surname from the forename, and chop each to ten characters. Both parts of JONATHAN MILKMAN then can be joined again. If the surname is long, as with SILAS PREPONDERANCE, then the name appears as SILAS PREPONDERA, which is enough to recognise it.

Now for some other string functions. Figure 4.2 shows a program that prints MSX COMPUTER as a title. What makes it more eyecatching is the fact that the word is printed with twelve asterisks on each side. The asterisks are produced by a string function whose instruction word is STRING$.

```
10 REM Remember CLEAR!
20 CLS
30 A$=STRING$(12,"*")
40 PRINTTAB(1)A$+"MSX COMPUTER"+A$
```

Fig. 4.2. Using concatenation to make a frame of asterisks for a title.

STRING$ means make a string out of, and it has to be followed by two items placed within brackets and separated by a comma. The first of these items is the number of identical characters that you want to put into this string. The second item is the character itself. In this example, we've used the * character, and it has had to be placed between quotes.

STRING$ is a useful way of creating strings of one character, and it's particularly useful when we come to look at graphics characters. There are, however, strings attached, as it were. One is string space. When your MSX computer is switched on, it reserves a small amount of memory for storing strings. The amount is fairly small, only enough for 200 characters, because a surprising number of programs will use less than this. When you use the STRING$ instruction a great deal, however, you can bite deeply into this small allocation, and this will cause your program to stop with an error message when the allocation is used up. The message is 'Out of string space in 30', and it requires you to reserve more space and try again. You can reserve more string space by the CLEAR instruction, which is hinted at in line 10 of Fig. 4.2. By using CLEAR 300, for example, we would reserve enough memory for 300 string characters. We don't need this much for this program, but it's as well to be on the safe side. Incidentally, this works both ways. If your program uses no string space at all, you could type CLEAR 0 at the beginning, and so get a little more memory space for other things.

There are two more points about the use of STRING$. The first is that you can't create a string of more than 255 characters, so the first number in a STRING$ expression has to be 255 or less. If you attempt to use a larger number you will get the 'Illegal function call' error message. The other point about STRING$ is that the second item in the brackets can be a number, with no quotes. Each character used by the MSX computer is represented by a code number, using what we call ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers. Figure 4.3

```
32        33  !   34  "   35  # 36  $

37  %     38  &   39  '   40  ( 41  )

42  *     43  +   44  ,   45  - 46  .

47  /     48  0   49  1   50  2 51  3

52  4     53  5   54  6   55  7 56  8

57  9     58  :   59  ;   60  < 61  =

62  >     63  ?   64  @   65  A 66  B

67  C     68  D   69  E   70  F 71  G

72  H     73  I   74  J   75  K 76  L

77  M     78  N   79  O   80  P 81  Q

82  R     83  S   84  T   85  U 86  V

87  W     88  X   89  Y   90  Z 91  [

92  \     93  ]   94  ^   95  _ 96  `

97  a     98  b   99  c   100  d 101  e

102  f   103  g  104  h   105  i 106  j

107  k   108  l  109  m   110  n 111  o

112  p   113  q  114  r   115  s 116  t

117  u   118  v  119  w   120  x 121  y

122  z   123  {  124  ¦   125  } 126  ~

127  ■
```

Fig. 4.3. The standard ASCII code numbers.

shows a printout of the ASCII code numbers and the characters that they produce on my printer (Epson RX-80). In place of the asterisk we used between quotes in Fig. 4.2, then, we could have used the number 42, making the instruction into STRING$(12,42), which is shorter.

The number characters of normal ASCII code extend only from 32 to

127. The code numbers above 127 are used by the MSX computer for other purposes, and we can select how we make use of them. Figure 4.4 gives a flavour of this; it is something that we'll investigate in much more detail in Chapter 7. By using the number 215 in the STRING$ command in line 20,

```
10 CLS
20 A$=STRING$(12,215)
30 PRINTTAB(1)A$+"MSX COMPUTER"+A$
```

Fig. 4.4. Using other ASCII codes.

we select a chequer pattern rather than a letter character. This same pattern can be typed by pressing the SHIFT, GRAPH and the P keys together. The effect is to produce a more effective looking frame for the name this time.

The long and the short of it

String variables allow us to carry out many operations that can't be done with number variables. One of these operations is finding out how many characters are contained in a string. Since a string can contain up to 255 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always a number so we can print it or assign it to a number variable. Since the number is *always* an integer, it should be assigned to an integer variable unless the program is a very short and simple one.

Figure 4.5 shows a useful example of LEN in use. This program uses LEN

```
10 T$="MSX Computing"
20 TB=(37-LEN(T$))/2
30 CLS:PRINTTAB(TB)T$
40 REM Now print your text.
```

Fig. 4.5. Using LEN to print titles centred.

as a way of printing a string called T$ centred on a line. This is an extremely useful routine to use in your own programs because it can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how any characters are present in the string T$. This number is subtracted from 37, and the result is then divided by two. If the number of characters in the string is an even number, the number TB will contain a .5, but this is completely ignored by TAB when the string is printed. Note how brackets have been used in line 20. The easiest way of writing a line like this is to start *at the innermost brackets*. For example, you know that you need to find the length of the string, T$, so you write

LEN(T$) first. You have to subtract this from 37, so you then add this item, to get 37 – LEN(T$). The *whole* of this, not just LEN(T$), must be divided by two, so you must place brackets around it, to get (37–LEN(T$))/2, which is then assigned to TB. You will find that this 'inside to outside' approach pays off when you have to work with lots of brackets. If you are uncertain about using brackets, be thankful that you are programming in BASIC, and not in the language called LISP! The whole process of centring could be done in one line, but I have shown it in three lines so that you can see the steps. In Chapter 6 we'll look at ways of rewriting actions like this so that they can be called up when we want them, just like another instruction word.

By the left, slice!

The next group of string operations that we're going to look at is called *slicing* operations. The result of slicing a string is another string – a piece copied from the longer string. Note that this is a *copying* process – nothing is removed from the longer string when the copy is made. The piece that is copied can be printed or assigned as you please. String slicing is a useful way of finding what letters or other characters are present at different places in a string.

All this might not sound terribly interesting, so take a look at Fig. 4.6. The

```
10 CLS
20 A$="Middlesex"
30 B$="Sugical"
40 C$="X-Ray Unit"
50 S$=SPACE$(1)
60 PRINTA$+S$+B$+S$+C$
70 PRINT:PRINTLEFT$(A$,1)+LEFT$(B$,1)
+LEFT$(C$,1)+" Computing."
```

Fig. 4.6. Using SPACE$ to make a space of the correct size, along with the string slicing action LEFT$.

strings A$, B$ and C$ are assigned in lines 20, 30 and 40. There's a new instruction in line 50, in the form of SPACE$. SPACE$ is a way of assigning a string which consists of spaces; as many as the number enclosed in the brackets. It's a simple and useful way of creating spaces, which saves having to use lines like:

```
100 SP$="   "
```

which are not easy to follow because you have to count the number of spaces for yourself. In this example, line 50 assigns just one space to the variable S$, so that we can use it to space words. Line 60 then prints a phrase on the screen and line 70 prints some slices from A$, B$ and C$ on the screen. Now how did the letters MSX appear? The instruction LEFT$ means copy part of a string starting at the left-hand side. LEFT$ has to be followed by two quantities within brackets and separated by a comma. The first of these quantities is the variable name for the string that we want to slice, A$ in the first example. The second is the number of characters that we want to slice (copy, in fact) from the left-hand side. The effect of LEFT$(A$,1) is therefore to copy the first letter from Middlesex, giving M. The next LEFT$(B$,1) copies the S from Surgical, and the last slice action of line 70 adds the X from X-Ray Unit. The last part of line 70 then adds the word Computing to these letters. How about trying for yourself a program which asks for your forename and surname, and then prints your initials?

You aren't confined to printing or assigning just one letter, of course. Suppose that you are working on a mailing list program for the local Darts Club. The names and addresses which are printed on the letters will be complete, but to save your typing finger(s), you want to be able to list the names on the screen using just the first five letters. You can then command the computer to find an address by just typing the first five letters of the surname. The part of this problem that we can solve easily now is the first five letters bit. Figure 4.7 shows how this is arranged. Line 20 asks for a

```
10 CLS
20 INPUT"Surname, please ";A$
30 B$=LEFT$(A$,5)
40 PRINT"Short form is ";B$
50 GOTO 20
```

Fig. 4.7. Entering a surname which is sliced to five letters. This action continues until you press CTRL STOP.

surname to be typed, and line 30 then uses LEFT$(A$,5) to copy the first five letters of the name. In this example no attempt has been made to do any more, and the program repeats endlessly because of the GOTO 20 in line 50. You will have to press CTRL STOP to make it halt. Later on, we'll look at ways of controlling this more effectively, so that all of the names can be kept in a list, and the first five letters copied as required. One thing at a time, if you please.

Eyes right

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful none the less. Figure 4.8 illustrates a simple use of this instruction to avoid having to use the whole of a complicated code number. Take a look, for example, at the code number on your telephone bill. There

```
10 CLS
20 READ D$
30 PRINT"Part No. is ";RIGHT$(D$,6)
40 DATA PD1Q-747-164027
```

Fig. 4.8. Using RIGHT$ to extract letters from the right-hand side of a string.

are other serious uses like this. You can, for example, extract the last four figures from a string of numbers like 010–242–7016. I said a string of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable you'll get a silly answer. Why? Because when you type N = 010–242–7016 the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N is −7248, which is not exactly what you had in mind! If you use N$="010–242–7016" then all is well.

Middle cut

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items within brackets, using commas to separate the items. Item 1 is the name of the string that you want to slice, as you might expect by now. The second item is a number which specifies where you want slicing to start. This number is the number of the characters counted from the left-hand side of the string, counting the first character as 1. The third item is another number; the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

It's a lot easier to see in action than to describe, so try the program in Fig. 4.9. Line 20 assigns A$ to the phrase Using Common putty. Line 30 then

```
10 CLS
20 A$="Using Common putty."
30 B$=MID$(A$,7,3)+MID$(A$,14,3)+MID$
(A$,3,3)
40 PRINTB$
```

Fig. 4.9. Using MID$, which can extract from any part of a string, and can, like LEFT$ and RIGHT$, be controlled by variables.

assigns a new string, B$, which is made out of slices from A$. The first slice uses MID$(A$,7,3). If you count the characters in A$, including spaces, you'll find that the seventh character is the C of Common. Remember that the counting for MID$ starts at 1, not at 0 like so many other counting actions. The slice starts with the C, and is of three characters – Com from this part of the phrase. The other two slices also take three characters each, to make up the word Computing, and this is what appears on the screen.

```
10 CLS
20 X=RND(-TIME)
30 INPUT"Your surname, please ";N$
40 L%=LEN(N$)
50 R%=RND(1)*L%+1
60 CD$=MID$(N$,R%,1)
70 PRINT"Your code letter is ";CD$
```

Fig. 4.10. Using a number expression along with slicing instructions.

One of the features of all these string slicing instructions is that we can use variable names or expressions in place of numbers. Fig. 4.10 shows a more elaborate piece of slicing, which uses an expression along with a random number. Line 20 ensures that the numbers are truly random, and the action all starts innocently enough in line 30 with a request for your surname. Whatever you type is assigned to variable N$, and in line 40 the length of this string is found and assigned to L%. Line 50 then generates a number, at random, which will lie between 1 and L%. We saw how this was done in Chapter 3, so the principles should be familiar by now. This random number, assigned to R%, will be a whole number because an integer variable can hold only a whole number. It is used in line 60 to select one of the letters from your name, and line 70 informs you that this letter is your code letter for today. It's a simple example, but the point is important – that whatever appears in the number part of MID$ (or LEFT$ or RIGHT$) can be a number variable. Could you now take this piece of program and alter it so that you get a group of letters of random length? The number of letters should not be more than half the number of letters of your name, so for SINCLAIR, I might get IN or CLA or INCL, for example.

Tying up more strings

It's time now to look at some other types of string functions, starting with two that are important when your program handles numbers. The first of these is VAL, and it's used to convert a number that is in string form back into ordinary number form so that we can carry out arithmetic. Suppose, for example, that we have NR$="3.4". NR$ is a string, and if we carry out PRINT NR$+"2" the result is 3.42, not 5.4. This is because numbers which are in string form *cannot* be added, and no other form of arithmetic is possible with them either. If you have a number in this form you can convert it by using VAL. You can, for example, use A!=VAL(NR$) to convert the number from its string form in NR$ to single-precision variable form as A!. As usual, you can choose whether to use an integer, single-precision or double-precision number variable. Remember that if you don't specify, the form will always be double-precision.

There's an instruction that performs the opposite conversion; STR$. When we follow STR$ by a number, number variable, or expression within

```
10 N$="22.5":V=2
20 CLS:PRINT
30 PRINTN$;" times ";V;" is ";V*VAL(N
$)
40 PRINT
50 V$=STR$(V)
60 PRINT"There are ";LEN(V$);" charac
ters in ";V$;"!"
70 PRINT
80 PRINTN$;" added to ";V$;" is not "
; N$+V$
```

Fig. 4.11. How VAL and STR$ are used to convert numbers to different forms.

brackets, we carry out a conversion to a string variable. We can then print this as a string, or assign it to a string variable name, or use string functions like LEN, MID$ and all the others. Figure 4.11 illustrates these processes – with a warning! Lines 10 to 30 show that we can perform arithmetic on N$ if we use VAL with it. Line 50 converts the number variable V into string form, using the string name of V$. Now V has been assigned to the number 2 in line 10, and we would expect just one character to be present in the string. Line 60 reveals that there are two! The reason is that when we use STR$ to convert a number into string form, a space is left at the left-hand side of the string in case we want to put in a sign (+ or −). This space is, of course, an invisible extra character, which explains why 2 appears to consist of two characters, and 42 of three characters. Line 80 shows the strings being concatenated, just to emphasise the difference between string variables and number variables.

The reason why

You may now be wondering why on earth we might want to use numbers in string form, when we have all this carry-on about converting between string form and number form. One reason is that string form is often very convenient. Just to give an example, you can enter *anything* in string form, using something like INPUT X$. If you have INPUT X, then what you enter *must* be a number, and only a number. You can't, for example, enter 27A. If you do, then you'll just get the usual 'Redo from start' message to remind you that only a number is acceptable, and 27A isn't an ordinary number. Now if only you will be using the program this might be acceptable, but if a non-programmer may use it this error message might cause a lot of confusion. If you use an input which is assigned to a string variable, then items like 27A will be accepted. You can then use VAL to extract the number part. This use of VAL, however, works *only if the string starts with a number*. You can extract the number 27 from 27A, but not from A27. If you type A27 as your answer, then VAL will give the number 0. When a reply consists of mixed numbers and letters, you will have to make use of MID$ or RIGHT$ to get rid of the letters before you use VAL. That's something that we'll come back to when we deal with loops in Chapter 5.

There's another reason for needing VAL. Up until now, we've used INPUT as our only way of getting a value into a program when it is running. INPUT, you remember, causes the program to hang up until you press the RETURN key. There's another way of getting a character from the keyboard, though, which uses a different instruction word; INKEY$. The important point about this one is that it has to be assigned to a string variable. You can use K$=INKEY$, but not K=INKEY$. The other point about INKEY$ is that it 'scans the keyboard'. This means that at the instant when the line that contains K$=INKEY$ is executed, the computer checks to find if any key is being pressed. If no key is pressed, the computer makes K$ equal to a blank string and goes on its merry way. If you want to make use of K$=INKEY$ to get something from the keyboard, then, you have to arrange for the instruction to be repeated until a key *is* pressed. Figure 4.12

```
10 CLS
20 PRINT"Press any key....."
30 K$=INKEY$:IF K$=""THEN 30
40 PRINT"Your key was ";K$
50 PRINT"Number value ";VAL(K$)
```

Fig. 4.12. The INKEY$ loop, which will always give a string. This can be converted by using VAL.

shows this in action. Line 30 assigns K$ to INKEY$, so that the computer will test the keyboard at this point. If K$ is a blank string we want this action to repeat, and this is done by the *test* which reads:

IF K$=""THEN 30

This means that if K$ is a blank string because no key was pressed, the next line should be line 30. This makes line 30 repeat until K$ is no longer a blank. Note how a blank string is typed as a pair of quotes with nothing between them – you just tap the quotes key twice to obtain this. The program will therefore hang up, repeating line 30, until you press a key. You need to press only one key, and unlike INPUT you don't have to follow it by pressing the RETURN key. Lines 40 and 50 then show the effect of what you have done. This is where VAL is really essential, because K$ is a string. If you want to use a number value here, then you can assign something like V%=VAL(K$). V% must be an integer, because INKEY$ allows you one key only, and one digit key can't give you a fraction!

Why should we need STR$? Let me give you just one example. Suppose you have a program which accepts numbers. These might be catalogue numbers of gifts, for example. Now when the computer lists are printed, we might want numbers like 1, 12, 123 to be printed out as 0001, 0012 and 0123 respectively, using four characters. It's quite difficult to arrange this if the

```
10 CLS
20 INPUT"Number, please ";V
30 V$=STR$(V)
40 L=LEN(V$)-1
50 V$="0000"+RIGHT$(V$,L)
60 V$=RIGHT$(V$,4)
70 PRINTV$
```

Fig. 4.13. Using STR$ to print numbers with leading (left hand) zeros.

program prints number variables, but it's simple if you use strings. Figure 4.13 illustrates what I mean. Line 20 obtains a number from you, and you should try the effect of numbers like 3, 45, 624, 1234 and so on, keeping to numbers of four digits or less. In line 30, the number is converted to string form as V$. Line 40 takes the length of this string and subtracts 1. This, remember, is because STR$ always places a blank space before the first digit of the number. We don't want this when we print the number, so we subtract 1 from L. The string is then concatenated with the string "0000" in line 50, and only the number part of V$ is added. This is done by using RIGHT$(V$,L). For example, if N=23, then LEN(V$) is 3, and L=2. Line 50 then takes the last two characters of V$, "23" and adds them to "0000", to get "000023". Line 60 then takes the last four digits of this, which are "0023", and this is printed in line 70. Note how a variable name like V$ can be *reassigned* several times in the course of a program like this. You could, of course, use different variable names at each stage of the process, but it's more economical to reassign the same name, and it makes things easier to follow. This is because you know that V$ is always being used to hold the quantity that you are working with. There are lots of other manipulations like this that become easy when STR$ is used. Another example would be adding letters to a number. Could you design part of a program that asks for your name and your age, then takes the first three letters of your name and joins them to your age to give a code like SIN52 ?

ASC and CHR$

If you look back to Fig. 4.3 now, you'll remember that we introduced the idea of ASCII code. This is the number code used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function ASC followed, within brackets, by a string character or a string variable. The result of ASC is a number; the ASCII code number for that character. If you use ASC("MSX"), you'll get the code for the M only, because the action of ASC includes rejecting more than one character. Figure 4.14 shows this in action. Line 20 asks you to press any key, and line 30 contains an INKEY$ to get the character from the keyboard. Line 40 then prints the ASCII code for whatever key has been pressed by using ASC(K$). When you run this you will find that keys which

```
10 CLS
20 PRINT"Press a key, please ";
30 K$=INKEY$:IF K$=""THEN 30
40 PRINT"ASCII code is ";ASC(K$)
50 GOTO 20
```

Fig. 4.14. Using ASC to find the ASCII code for letters.

don't produce anything on the screen will still give an ASCII code. Keys such as the spacebar, the ESC, TAB and HOME keys, for example, all give their own codes. You will also find that the CTRL key gives no code of its own, but when it is pressed along with another key a new code is generated. Try the effect of SHIFT and CODE along with letter keys as well.

ASC has an opposite function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. Figure 4.15 is a short program that allows you to enter numbers and

```
10 CLS
20 LOCATE2,10
30 INPUT"Number, please ";N
40 PRINT:PRINT"Character is ";CHR$(N)
50 PRINT:PRINT"Press any key to proce
ed"
60 K$=INKEY$:IF K$=""THEN 60
70 GOTO10
```

Fig. 4.15. Using CHR$ to find what character shape corresponds to a number code.

see what their effect is on the screen. The numbers that can be used for this CHR$ action extend from 0 to 255. The numbers from 0 to 31 do not produce any visible character on the screen. These are 'action' code numbers, which produce effects like backspacing the cursor, clearing the screen and so on. Figure 4.16 lists these effects. The number 32 is the ASCII code for the spacebar, and the numbers from 33 to 255 will all produce various characters.

One of the main uses of CHR$, which we shall investigate in Chapter 7, is in producing graphics shapes, the other is for coding messages. Every now and again it's useful to be able to hide a message in a program so that it's not too obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. The codes can be kept in a DATA line, read in one by one, converted to characters by using CHR$, and printed. This is something that we'll look at when we come to the subject of loops in the next chapter.

Code	Effect
1	Make next character a graphic.
2	Move cursor to first character of word to the left.
3	Stop program.
4	Nil.
5	Delete rest of line.
6	Move cursor to first character of next word.
7	Sound beep.
8	Backspace cursor by one step and delete character.
9	Move cursor eight spaces to the right.
10	Move cursor to first position on next line.
11	Move cursor to top left corner of screen.
12	Clear the screen.
13	As for RETURN key.
14	Cursor to end of line.
15–17	Nil.
18	Insert character at cursor position.
19,20	Nil.
21	Delete line.
22–27	Nil.
28	Cursor right.
29	Cursor left.
30	Cursor up.
31	Cursor down.

Fig. 4.16. The effects of the ASCII codes 0 to 31.

The law about order

We saw earlier in Fig. 3.9, and we'll look again in Fig. 5.12, how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign = means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to compare complete words, character by character. Figure 4.17 illustrates this use of comparison using the = and > symbols. Line 20 assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a

```
10 CLS
20 A$="QWERTY"
30 PRINT:INPUT"Type a word (capitals)
 ";B$
40 IF B$=A$ THEN PRINT "Same as mine!
":END
50 IF A$>B$ THEN SWAP A$,B$
60 PRINT"Correct order is ";A$;" then
 ";B$
70 END
```

Fig. 4.17. Comparing words to decide on their alphabetical order.

word, using upper-case (capital) letters. The comparisons are then carried out in lines 40 and 50. If the word that you have typed, which is assigned to B$, is *identical* to QWERTY, then the message in line 40 is printed and the program ends. If QWERTY would come later in an index than your word, then line 50 is carried out. If, for example, you typed PERIPHERAL, then since Q comes after P in the alphabet and has an ASCII code that is greater than the code for P, your word B$ scores lower than A$, and line 50 swaps them round. MSX computers do this by using the useful command SWAP. When SWAP is followed by two variable names, separated by a comma, it will do what the name suggests – swap the values. This is the command that has been used in line 50 following the IF test. Line 60 will then print the words in the order A$ and then B$, which will be the correct alphabetical order. If the word that you typed comes later than QWERTY, for example TAPE, then A$ is not 'greater than' B$, and the test in line 50 fails. No swap is made, and the order A$, then B$, is still correct. Note the important point though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts. The SWAP command applies to number variables as well as to string variables.

String arrays

We looked briefly at the idea of number arrays in Chapter 3, showing how a single variable name could be used for a set of numbers. The same arrangement can be used for strings, and the only change that needs to be made is that a string variable name has to be used. We can, for example, use A$(1), A$(2), A$(3) and so on, to represent a set of strings, which might be the names of the members of the local Music Society (or the violinists, perhaps). A string array like this also has to be dimensioned so that the computer can set aside memory for storing the strings. If, for example, you want to use the array A$ for up to item A$(100), then you need a line that reads: DIM A$(100). This line would have to be carried out *before* you start assigning values to these A$ elements. If you find in the course of a program that you need more array items and you haven't dimensioned enough, hard luck! You cannot dimension the same array a second time while the program is running. If your early line was DIM A$(100), you cannot have a later line of DIM A$(200). This is because the computer has set aside memory for the first dimensioning, and will have made use of the memory around this reserved piece. Attempting to re-dimension A$ would cause the computer to clear some of the other parts of its memory, and this could result in the program being completely destroyed. When the computer comes across a second DIM statement about the same variable name, then, you get a 'Re-dimensioned array' error message, and the program stops. The MSX machines allow you to use subscript numbers up to 10 *without* needing to use DIM. This allows you eleven items, because A$(0) can be used as well as A$(1) and so on. What you have to watch if you take advantage of this is that you do not use an array like this and *then* try to dimension it by a command like DIM A$(20). This also will cause the 'Re-dimensioned array' message. We'll look at examples of string arrays in use in Chapter 5.

String ends

There are, inevitably, a few commands that we haven't looked at yet, mainly because they haven't fitted in with the others. One of these is the interesting and useful one, INPUT$. This is not quite like INPUT, because it allows you to enter a preset number of characters, and they are not shown on the screen when you enter them. It's ideal for security codes, as Fig. 4.18 shows. Line 30

```
10 CLS
20 PRINT"Please type the 5-letter cod
e"
30 A$=INPUT$(5)
40 IF A$<>"QSRBN"THEN PRINT "Incorrec
t- no entry":GOTO10
50 PRINT"Pass, friend."
```

Fig. 4.18. How INPUT$ is used for 'security entry'.

contains the step A$=INPUT$(5). This means that only five characters can be accepted for this input, and the string of characters will be assigned to A$. The computer hangs up and waits for you when line 30 runs, but you *don't* need to press ENTER. Immediately you press the fifth key, the entry is complete, but without anything appearing on the screen. In this example, line 40 then checks that what you have entered is the correct password. It's a very useful way of getting an entry of the right number of characters. There's no need to count characters and use a test to detect the entry of the wrong number of characters. An extension to INPUT$ allows entry from tape or disk.

Another pair of useful commands uses FRE. If you type PRINT

FRE(A$), then the machine will print the number of bytes of memory that you can use for strings. You can use any variable name in place of A$, and it doesn't need to be a variable name that is used or assigned in your program. A variable name used in this way is called a *dummy variable*. You will find that when you switch on the machine, 200 bytes of memory are reserved for strings; space for 200 characters. This isn't a lot, but a surprising number of programs use even less than this. You already know how to extend the string space using CLEAR. If you use a dummy *number* variable with FRE, as for example PRINT FRE(A), then the computer will print the *total* amount of memory that is available. This may give you a nasty shock if you thought that your machine had 64K (=65536 bytes) of free memory! You can use FRE(A) to decide if you have to stop using a program because of lack of memory. By having a line such as:

```
IF FRE(A)<1000 THEN PRINT "No room - please record
data":GOSUB 5000
```

you can detect when memory is running short, and then record your data.

Finally, MSX machines can make use of an excellent command, INSTR. This is used to find if one string is contained in another. It's used in the simple form:

```
X%= INSTR (A$, B$)
```

to find if B$ is contained in A$. If it is, then X% is the position number of the first letter of B$ that is found in A$. If B$ is *not* contained in A$, then X% is zero. X% will always be zero if B$ is longer than A$. You can, of course, use the form:

```
PRINT INSTR(A$,B$)
```

if you just want to see the number.

Figure 4.19 shows a simple example of this function in action. Lines 20 to 40 allocate names to strings, and lines 50 to 70 make the tests, so that you can see how they work out. Notice that the strings have to be exact for the function to work – it's no good looking for Bert if what is contained in the

```
10 CLS
20 A$="Albert Hall"
30 B$="Richardson, Bertram"
40 C$="Sinclair, I"
50 PRINT"In ";A$;" bert is located at
  ";INSTR(A$,"bert")
60 PRINT"In ";B$;" Bert is located at
  ";INSTR(B$,"Bert")
70 PRINT"In ";C$;" BERT is located at
  ";INSTR(C$,"BERT")
```

Fig. 4.19. Using INSTR to find if a group of letters is contained within another group.

string is bert or BERT, for example. To leave you with a thought, suppose you had a string A$="YESyesYUPyupSUREsureOKok", and you asked for a yes/no answer. You could get INSTR to look through this. If the result of X%=INSTR(Answer$,A$) is zero, then the answer wasn't any form of YES! The other point about INSTR is that you can specify at which character in the string you start the search. This is done by putting in a number as the first item within the brackets. The other items are used as before, with commas between them. For example, if you had:

```
    X%=INSTR(5,A$,B$)
```

the computer would start at character number 5 of A$, and look from that position to find if B$ was present. This can be useful if, for example, you are looking for a space between a forename and a surname. The program might be misled if there was a space just before the name, so using INSTR(2,A$,B$) would skip over this first space and concentrate on looking for the second one. The number X% that you obtain from this can then be used in MID$, LEFT$ or RIGHT$ commands to separate out the words. Magic!

Chapter Five
Repeating Yourself

One of the activities for which a computer is particularly well suited is repeating a set of instructions, and every computer is well equipped with keywords that will cause repetition. The MSX computers are no exception to this rule. We'll start with the simplest of these 'repeater' actions, one which we have already used, GOTO.

GOTO means exactly what you would expect it to mean – go to another line number. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again.

Figure 5.1 shows an example of a very simple repetition or 'loop', as we

```
10 PRINT"MSX COMPUTING FILLS YOUR SCR
EEN"
20 GOTO10
```

Fig. 5.1. A very simple loop. You can stop this by pressing the CTRL and STOP keys.

call it. Line 10 contains a simple PRINT instruction. When line 10 has been carried out, the program moves on to line 20, which instructs it to go back to line 10 again. This is a never-ending loop, and it will cause the screen to fill with the words:

MSX COMPUTING FILLS YOUR SCREEN

until you press the CTRL and STOP keys to 'break the loop'. Any loop that appears to be running forever can be stopped by pressing the STOP key. This does what it says, stops the program running, but not completely. If you press the STOP key again, the program will take over from where it left off. We'll see later that this is very useful if you are chasing faults in a program. If you want to stop the program completely, so that you can record it or change it, then you have to press the CTRL key and the STOP key together.

```
10 CLS:N=0
20 PRINT N
30 N=N+1
40 GOTO 20
50 REM Use CTRL and STOP again.
```

Fig. 5.2. A loop which carries out a count-up action very rapidly. You will also have to use the CTRL and STOP keys to stop this one.

Now try a loop in which there is slightly more noticeable activity. Figure 5.2 shows a loop in which a different number is printed out each time the computer goes through the actions of the loop. We call this 'each pass through the loop'. Line 10 sets the value of the variable N at 0. This is printed in line 20, and then line 30 increments the value of N. Line 40 forms the loop, so that the program will cause a very rapid count-up to appear on the screen. Once again, you'll have to use the STOP key to stop it, and this gives you a chance to see how the program will carry on the next time that you press the STOP key. As before, pressing CTRL and STOP together will break out of the program.

Now an uncontrolled loop like this is not exactly good to have, and GOTO is a method of creating loops that we prefer not to use! We don't always have an alternative, but there is one – the FOR...NEXT loop. As the name suggests, this makes use of two new instruction words, FOR and NEXT. The instructions that are repeated are the instructions that are placed between FOR and NEXT. Figure 5.3 illustrates a very simple

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"MSX COMPUTERS RULE O.K."
40 NEXT
```

Fig. 5.3. Using the FOR...NEXT loop for a counted number of repetitions.

example of the FOR...NEXT loop in action. The line which contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 10. The NEXT is in line 40, and so anything between lines 20 and 40 will be repeated.

As it happens, what lies between these lines is simply the PRINT instruction, and the effect of the program will be to print MSX COMPUTERS RULE O.K. ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the NEXT instruction is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then line 30 is repeated, and this will continue until the value of N exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

You don't have to confine this action to single loops either. Figure 5.4

```
10 CLS
20 FOR N=1 TO 10
30 PRINT"Count is ";N
40 FOR J=1 TO 500:NEXT
50 CLS:NEXT
```

Fig. 5.4. A program that uses nested loops, with one loop inside another. The inner loop is a *delay loop*.

shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one. When loops are nested in this way, we can describe the loops as 'inner' and 'outer'. The outer loop starts in line 20, using variable N which goes from 1 to 10 in value. Line 30 is part of this outer loop, printing the value that the counter variable N has reached. Line 40, however, is another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is clearing the screen in line 50. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used NEXT to indicate the end of each loop. We could use NEXT J in line 40 and NEXT N in line 50 if we liked, but this is not essential. It also has the effect of slowing the computer down, though the effect is not important in this program. When you do use NEXT J and NEXT N, you must be absolutely sure that you have put the correct variable names following each NEXT. If you don't, the computer will stop with a NEXT without FOR error – meaning that the NEXTs don't match up with the FORs in this case. You would also get this message if you had omitted a NEXT.

Even at this stage it's possible to see how useful this FOR...NEXT loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N=1 TO 9 STEP 2
```

which would cause the values of N to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of 1.

Figure 5.5 illustrates an outer loop which has a step of −1, so that the count is *downwards*. N starts with a value of 10, and is decremented on each pass through the loop. Line 40 once again forms a time delay so that the count-down takes place at a civilised speed. This is a particularly useful way of slowing the count-down. If we want to speed the rate up, the easiest way is

```
10 CLS
20 FOR N=10 TO 1 STEP -1
30 PRINT N;" seconds and counting."
40 FOR J=1 TO 500:NEXT
50 CLS:NEXT
60 PRINT"BLASTOFF!"
```

Fig. 5.5. A count-down program, making use of STEP.

to use an integer variable such as N% in place of N. If we do this, however, we can't use steps that contain fractions, like .1.

Every now and again, when we are using loops, we find that we need to use the value of N (or whatever variable name we have used) after the loop has finished. It's important to know what this will be, however, and Fig. 5.6

```
10 CLS
20 FOR N=1 TO 5
30 PRINT N
40 NEXT
50 PRINT "N is now ";N
60 FOR N=5 TO 1 STEP -1
70 PRINT N
80 NEXT
90 PRINT "N is now ";N
```

Fig. 5.6. Finding the value of the loop variable after a loop action is completed.

brings it home. This contains two loops, one counting up, the other counting down. At the end of each loop, the value of the counter variable is printed. This reveals that the value of N is 6 in line 50, after completing the FOR N = 1 TO 5 loop, and is 0 in line 90 after completing the FOR N = 5 TO 1 STEP −1 loop. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have changed by one more step at the end of the loop.

One of the most valuable features of the FOR...NEXT loop, however, is the way in which it can be used with number variables instead of just numbers. Figure 5.7 illustrates this in a simple way. The letters A, B and C are assigned as numbers in the usual way in line 20, but they are then used in a FOR...NEXT loop in line 30. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have anything that represents a number or can be worked out to give a number, then you can use it in a loop like this.

```
10 CLS
20 A=2:B=5:C=10
30 FOR N=A TO B STEP B/C
40 PRINT N
50 NEXT
```

Fig. 5.7. A loop instruction that is formed with *number variables*.

Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is to enter numbers while the computer keeps a running total, adding each number to the total of the numbers so far. From what we have done so far, it's easy to see how this could be done if we wanted to use numbers in fixed quantities, like ten numbers in a set. The program of Fig. 5.8 does just this.

```
10 TT=0:CLS
20 PRINTTAB(6)"Totalling Numbers Prog
ram"
30 PRINT:PRINT"Enter each number as r
equested."
40 PRINT"The program will give the to
tal."
50 FOR N=1 TO 10
60 PRINT"Number ";N; "please ";
70 INPUT J:TT=TT+J
80 PRINT"Total so far is";TT
90 NEXT
```

Fig. 5.8. A number-totalling program for ten numbers.

The program starts by setting a number variable TT to zero. This is the number variable that will be used to hold the total, and it has to start at zero. As it happens, the MSX computer arranges this automatically at the start of a program, but it's a good habit to ensure that everything that has to start with some value actually does. We can't, incidentally, use TO for this variable, because TO is a reserved word, part of the FOR...NEXT set of words. You will get a 'Syntax error' message when the program runs if you have used a 'reserved word' as a variable name.

Lines 20 to 40 issue instructions, and the action starts in line 50. This is the start of a FOR...NEXT loop which will repeat the actions of lines 60 to 80 ten times. Line 60 reminds you of how many numbers you have entered by printing the value of N each time, and line 70 allows you to INPUT a number which is then assigned to variable name J. This is then added to the total in the second half of line 70, and line 80 prints the value of this total. The loop then repeats. At the end of the program, the final total has been printed.

It's all good stuff, but how many times would you want to have just ten numbers? It would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a *terminator*, something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference. To do this, we have to

```
10 TT=0:CLS
20 PRINTTAB(6)"Totalling Numbers Prog
ram"
30 PRINT:PRINT"Enter each number as r
equested."
40 PRINT"The program will give the to
tal."
50 FOR N=1 TO 100
60 PRINT"Number ";N; "please ";
70 INPUT J:TT=TT+J
80 IF J=0 THEN N=100
90 PRINT"Total so far is";TT
100 NEXT
```

Fig. 5.9. How to break out of a FOR...NEXT loop if you want to.

test the number that is input, and change the action of the loop if the input is 0. Figure 5.9 shows one way of doing this. The program is very much as before, but a new line 80 has been added. This uses the keyword IF to make the test: IF J=0 THEN N=100. What this amounts to is that if the number which was entered in line 70 was 0, then the counter number of the FOR...NEXT loop, N, is set to its final value. This will stop the loop, and so stop this program.

You might wonder why we don't just make line 80 read: IF J=0 THEN 200 and place a line 200 END at the end of the program. The answer is that you can, in a simple example like this, and it works. In a longer and more complicated program, though, jumping out of a loop in this way can cause trouble. The trouble manifests itself in the form of the program suddenly going haywire at some later time, usually after you have entered a lot of data and taken a lot of time over it. The principle that is illustrated in Figure 5.9 is the safe way of ending a FOR...NEXT loop before the normal limit of loops.

There are other ways, however, and Fig. 5.10 shows an example of one of

```
10 CLS:PRINTTAB(12)"Running Total"
20 PRINT:PRINT"The program will total
 numbers for":PRINT"you."
30 PRINT"Enter 0 to stop."
40 TT=0:N=0
50 N=N+1:PRINT"Item ";N;" is ";
60 INPUT J
70 IF J=0 THEN 110
80 TT=TT+J
90 PRINT"Total is ";TT
100 GOTO50
110 PRINT"Final total is ";TT
```

Fig. 5.10. A running total program which doesn't use FOR...NEXT. The number is tested near the start of the loop.

them in action. We don't use a FOR...NEXT loop, because we don't know in advance how many times we might want to go through the loop, so we have to go back to using GOTO. This time, however, we'll keep GOTO under closer control – the word won't even appear in the program! This time the instructions appear first, but we still have to make the total variable TT equal to zero in line 40. In the same line, a variable N is also set to zero. Line 50 increments the value of N, so that when line 50 runs for the first time, it prints:

Item 1 is ?

and waits for you to type the number and press RETURN. Each time you type a number, then, in response to the request in line 50, the number that you type is tested in line 70. If the number is zero, then the program jumps to line 110, where the final total is printed, and the program ends. If the number is not zero, though, it is added to the total in line 80, and line 90 prints the running total. In line 100, the program is forced to return to line 50 for the next number entry.

A loop of this type is called a WHILE...DO loop, and some computers allow you to make the loop using these words instead of using GOTO. The reason for the name is that while J is not zero, the loop does the totalling action. The test is made *before* the number is added. When we use 0 to terminate the loop, this is not important, but if we were using a number such as −1, then it would be important not to add in this value.

There is another form of loop, called the REPEAT...UNTIL loop. Some computers allow these words to be used but, on MSX machines, we once again have to use GOTO (or THEN) to form the loop. An example is shown in Fig. 5.11. In this one, the total variable TT is set to zero in line 40, and then

```
10 CLS:PRINTTAB(12)"Running Total"
20 PRINT:PRINT"The program will total
 numbers for":PRINT"you."
30 PRINT"Enter 0 to stop."
40 TT=0
50 INPUT"Number, please ";J
60 TT=TT+J
70 PRINT"Total so far is ";TT
80 IF J<>0 THEN 50
90 PRINT"End of totalling."
```

Fig. 5.11. Another running total loop, with the number tested near the end of the loop.

line 50 gets the input number and assigns it to variable J. This is added to the total in line 60, and line 70 prints the value of the total so far. Line 80 is the loop controller, with the IF test. The test in line 80 is to see if the value of N is *not* equal to zero. The odd-looking sign that is made by combining the less than and the greater than signs, <>, is used to mean not equal, so the line reads: 'if N is not equal to zero, then (GOTO) line 50'. We can put the GOTO in, or leave it out. Since it's just a few more letters to type, I've left it out.

The effect, then, is that if the number which you typed in line 50 was not a zero, line 80 will send the program back to repeat line 50. This will continue until you do enter a zero. When this happens, the test in line 80 fails (N is zero), and the program looks for a line 90. This line announces the end of the program, and since there are no more lines, the program stops. When this type of loop is used, the actions of the loop will always run at least once, because the test is placed at the end of the loop. There's just one thing that you have to be careful about in programs of this type. When the program starts, enter a number, say 2. From then on, don't press a number key, just the ENTER key. You'll see the number 2 entered automatically each time! This is because the machine keeps each INPUT in a special piece of memory, and it's only altered by another INPUT. You have to be careful with totalling programs because of this; if no key is pressed, then pressing ENTER will still have an effect – it will enter the previous number once again. Getting round this one is not quite so easy. You have to write a loop which contains INKEY$, and which adds characters to a variable name until RETURN is pressed. You'll know how to do that by the time you finish this book. These types of loops allow you much more freedom than a FOR...NEXT loop, because you are not confined to a fixed number of repetitions. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

Decisions, decisions

We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 5.12. The mathematical signs are used for convenience, and you have to remember which way round the

Sign	Meaning
=	Quantities are identical.
>	Quantity on left is greater than quantity on right.
<	Quantity on left is less than quantity on right.
⩾	Quantity on left is greater than or equal to quantity on right.
⩽	Quantity on left is less than or equal to quantity on right.
<>	Quantities are not equal.

Fig. 5.12. The mathematical signs used for comparing numbers and number variables.

greater than and less than signs have to be. It's important to note that the equals sign means identical to when it is used in a test like this. If A is 4.9999999 and B is 5.0000000 then a test such as IF A = B will fail. A is not identical to B, even though it is close enough to be equal to our eyes. The important point here is that the numbers we see on the screen have been rounded, so that PRINT A in the example above might give the result 5. The test, however, is made on the numbers which have not been rounded.

Figure 5.13 shows another test – this time on string variables. The instruction is in line 20, you are asked to type the y or n key. Line 30 gets

```
10 CLS
20 PRINT"Type y or n"
30 K$=INKEY$:IF K$=""THEN 30
40 IF K$="y" THEN 100 ELSE IF K$="n"
THEN 200
50 PRINT"Your answer ";K$;" is not y
or n...":PRINT"Please try again."
60 GOTO 30
70 END
100 PRINT"That was y for YES.":END
200 PRINT"That was n for NO":END
```

Fig. 5.13. Testing string variables, in this example to find whether a reply is y or n. ELSE has been used to provide a mugtrap.

your answer; you have only to press the y or n key without touching RETURN. The key that you have pressed has its value assigned to K$, so that K$ should be y or n. Line 40 then analyses this result. If the key that you pressed was neither y nor n, then the program ignores the THEN 100 and THEN 200 instructions of line 40, and goes on to line 50 and 60. This tells you that you didn't press either y or n, and you must try again. A line like this is called a *mugtrap*.

The tests in line 40 of this example are for identity. Only if K$ is absolutely identical to y will the program jump to line 100, and print the phrase: That was y for YES. Using INKEY$ in place of INPUT does not allow you to make such mistakes as typing a space ahead of y, or a space following it. You could, of course, type Y in place of y, in which case K$ will *not* be identical, and the test fails. If the first test failed, however, then ELSE forces the second test to be tried. This time, the answer is tested for the letter n and if this is found the program jumps to line 200. This line then prints: That was n for NO, and the program ends once again. It's up to you to form these tests so that they behave in the way that you want! You can use AND and OR to make the tests apply to more than one thing, so you can use IF K$=Y OR K$=y, for example, to test either form of the Y key.

The MSX computer is one of a select group of computers that allows you to use the instruction word ELSE, and it offers an alternative to the test that is carried out by IF. In the example of Fig. 5.13, two tests were combined.

You can, however, combine much more than this. You can use lines like IF X=3 THEN 100 ELSE IF X=4 THEN 200 ELSE IF X=5 THEN 300 ... and so on. When lines get as complicated as this, though, they become hard to follow, and there are easier ways of achieving the same effect as we shall see.

Looping to a purpose

So far, we have been looking at short examples of loops which were designed to show how loops are constructed. It's time now to look at examples of loops in use, and to see how a program which includes a loop is designed. All loops are intended to carry out a set of actions over and over again. What you have to decide before you try to write the BASIC of a looping program is what actions you want to repeat, and what will make the loop stop. If it is possible to design the loop so that it repeats some definite number of times, this should be done. The reason is that this would allow you to use a FOR...NEXT loop, rather than trying to make up a loop with GOTO. The trouble with GOTO loops is twofold. First of all, the start of the loop is not marked. When you read a program listing, you can see where a FOR...NEXT loop starts – in the line which contains FOR. You don't know where a GOTO loop starts, because the only thing that indicates it is the line number that follows GOTO (or THEN). If you see a line that reads:

```
200 GOTO 100
```

then it's a fair bet that there is a loop that starts at line 100, but you have had to read a lot of the program to find it! The other difficulty about GOTO loops is that it's very easy to make a mistake and go to the wrong line. The result might be a program that doesn't work. Even worse, the result can be a program that looks as if it works, but doesn't give the correct results.

We'll look, then, at a very simple number-guessing game and how it is designed. The listing is shown in Fig. 5.15. Fig. 5.14 shows the plan which was used to design it. This plan consists of a set of steps, with brackets used to expand some steps into more detail. The description contains no keywords and, in fact, it should not because the use of keywords makes it more difficult to follow. The plan starts with the words Ten times, to show that we want the steps of the program to be repeated this number of times. This allows us to make use of a FOR...NEXT loop, which is the best type of loop to use in MSX BASIC. The next step is selecting a number at random. This is the first of the steps that will be repeated ten times, and it is followed by Input guess. This is where the user of the program enters the number that is guessed. The next steps are concerned with scoring. If the guess is exactly correct, then two points are scored, and the arrow shows that the next step must be the pause. As an alternative, if the guess is close, one point is added to the score, and again the program moves to the pause stage. If the guess is

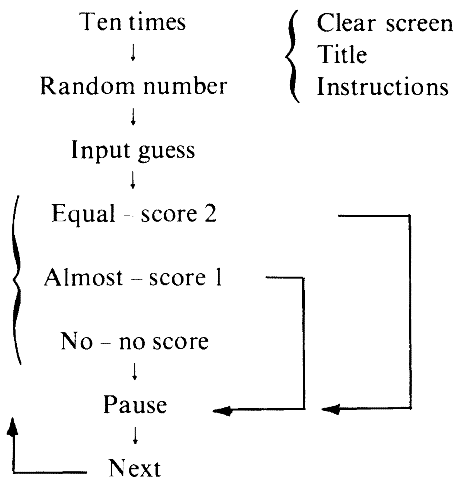


Fig. 5.14. The design steps for the number-guessing game in Fig. 5.15.

completely out, then some message will be printed ('No score', perhaps), and once again, the program pauses. The pause will be about a couple of seconds, and after the pause, the program moves to repeat the loop.

The next step is to fill in some details. This is done on the right-hand side of the set of steps of the plan, using brackets to show where several more detailed steps have to be inserted. The points that have been put in here are where the CLS, heading, and instructions steps are placed. We also make notes about messages, and the length of time of the pause. For a simple program like this, that's all we need to start writing the BASIC lines. You don't necessarily have to write *numbered* lines yet, though. At this stage, it makes more sense to write BASIC for one step at a time, and the order of running the steps is not usually the best order for writing. For example, there isn't much point in writing a heading and instructions until we're sure that the program works. The steps that we should concentrate on first are the selection of a random number and the scoring steps because, unless these are correct, the rest of the program is of little use.

Start with the random number, then. Since we are dealing with numbers that will all be integers, we can assign an integer dealing with numbers that will all be integers, we can assign an integer variable, and use:

```
X%=INT(RND(1)*10+1)
```

to get a random number as X%. This, remember, will get a random number which lies between 1 and 10. To make sure that the sequence of numbers is different each time the program runs, we will have to use X%=RND (−TIME) early in the program *before* the loop starts. Make a note of it! The testing for equality is easy enough, and we can settle a variable name for the guessed number – N%. There will be a GOTO at the end of this line to lead to the Pause step. Testing for near equality can be done by using ABS(N%–X%). ABS will make whatever lies between the brackets into a positive quantity, so if N% happens to be less than X%, the result will be the difference, but with a positive sign. Once again, this step has to be ended with a GOTO to make the Pause step come next.

We can then look at the Pause step. MSX BASIC provides for a variable which is called TIME. This can be assigned like any other variable, but its value is incremented 50 times per second (60 times per second in the USA). This incrementing action is completely automatic, and needs no attention. If we set TIME=0, and in the next line keep testing to find when TIME exceeds 100, we shall have achieved a two-second pause (in the European version). Now there's a last minute thought. It looks odd to have the bottom line of the screen always showing the function key words while this program is running. We can shut off this display when our program starts by using KEY OFF. At the end of the program, we can restore the Key display by using KEY ON. Now we need only make a few notes at the side of the plan about where on the screen we want the messages to appear and we're ready to write the final version in Fig. 5.15.

Line 10 in Fig. 5.15 switches off the KEY display, sets the score variable SC% to zero, carries out the RND(–TIME) step, and starts the loop.

```
10 KEY OFF:SC%=0:X%=RND(-TIME):FOR J%
=1 TO 10
20 CLS:PRINTTAB(10)"GUESS THE NUMBER"
30 PRINT:PRINT"If you get near, I'll
tell you":PRINT"Number is between 1 a
nd 10."
40 PRINT:PRINT"Attempt ";J%:PRINT
50 X%=INT(RND(1)*10+1)
60 INPUT"YOUR GUESS - ";N%
70 IF N%=X% THEN PRINT"Spot on. Score
 2":SC%=SC%+2:GOTO 100
80 IF ABS(N%-X%)<3 THEN PRINT"Near- i
t was ";X%;" .Score 1":SC%=SC%+1:GOTO
100
90 PRINT:PRINT"No score."
100 LOCATE 9,20:PRINT"SCORE TOTAL IS
";SC%
110 TIME=0
120 IF TIME<100 THEN 120
130 NEXT
140 KEY ON
```

Fig. 5.15. A simple number-guessing game which uses number comparisons.

Because the FOR J%=1 to 10 step is the last one in this line, the first three steps are *not* repeated on each pass through the loop. The steps of the loop start with line 20. This clears the screen, and prints the heading. Line 30 then provides brief instructions. Line 40 prints the attempt number, so that the user knows how many shots have been used up.

The real action starts in line 50, where the X%= INT(RND(1)*10+1) step causes variable X% to take a whole-number value that lies between 1 and 10. You enter your number at line 60, and the tests are made in lines 70 and 80. If the number that you picked is identical to the random number, then you get the 'Spot on' message in line 70, two points are added to the score variable SC%, and the GOTO 100 skips over the other tests to get to the Pause stage. The less obvious test is in line 80. If the difference between your guess and the actual number is less than 3 (meaning 1 or 2) then the message in line 80 is printed, the score is bumped up by one point, and you move to the Pause. If you don't get anywhere near, the program moves to line 90 to announce 'No score'. The pause is then carried out, using TIME, and then line 130 contains the NEXT that will make the loop repeat. It's very simple, but quite effective.

Many of the commands that we have looked at in previous chapters take on much more meaning when we carry them out inside loops. This is particularly true when the counter variable of the loop can be used as part of the action. Take a look at Fig. 5.16, for example, which makes use of the fact

```
10 CLS
20 INPUT"Your name, please ";NM$
30 L%=LEN(NM$):C%=(L%/2)+1
40 FOR N%=1 TO C%
50 PRINTTAB(21-N%)MID$(NM$,C%-N%+1,N%
*2-1)
60 NEXT
```

Fig. 5.16. Using loop variables to make a letter pyramid to show the action of MID$ with a formula.

that we can use variable names or expressions in place of numbers in string-slicing actions. It all starts innocently enough in line 20 with a request for your name. Whatever you type is assigned to variable NM$, and in line 30 a bit of mathematical juggling is carried out. How does it work? Suppose you type as your name DONALD. This has six letters, so in line 30, L% is assigned to 6, and C% is the whole number part of L%/2 (equal to 3), plus 1, making 4. Line 40 then starts a loop of 4 passes. In the first pass you print at TAB(20) because N%=1. What you print is the MID$ of the name using C%−N%+1, which is 4−1+1=4, and N%*2−1, which is also 1. What you print is therefore MID$(NM$,4,1), which is A in this example. On the next run through the loop, N% is 2, C%−N%+1 is 3, and N%*2−1 is also 3. What is printed in MID$(NM$,3,3), which is NAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name! If your name is short, try making up a longer one. Could you try designing a variation on this which worked in the opposite way, starting with the full name and cutting a letter off each side on each pass through the loop?

We looked briefly in Chapter 4 at the idea of coding messages in ASCII, and reading them from data lines. Figure 5.17 illustrates this use. Line 50 contains an INKEY$ loop to make the program wait for you. When you

```
10 CLS:PRINT
20 PRINT"What does MSX mean?"
30 PRINT
40 PRINT"Press any key to find out"
50 K$=INKEY$:IF K$=""THEN 50
60 PRINT
70 FOR J%=1 TO 23:READ N%
80 PRINTCHR$(N%);
90 NEXT
100 END
110 DATA77,97,114,118,101,108,108,111
,117,115,32,83,108,105,99,107,32,88,9
7,109,112,108,101
```

Fig. 5.17. Using ASCII codes to carry a coded message, and then using CHR$ in a loop to obtain the character that corresponds to a code number.

press a key, the loop that starts in line 70 prints 23 characters on the screen. Each of these is read as an ASCII code from a list, using a READ...DATA instruction in the loop. The PRINT CHR$(N%) in line 80 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent de-coders! This example, incidentally, illustrates the use of READ and DATA in a loop. We would normally use READ and DATA only for information that we particularly wanted to keep stored in a program like this.

While we are on the subject of READ and DATA, there's another twist to this instruction in the form of RESTORE. RESTORE means that the DATA list will start again from the beginning. If you READ all of the data, and then want to read it all again, you will have to have a RESTORE instruction before the second READ loop. If you didn't, you would get an 'Out of data' error message. RESTORE, however, can do more than this. Take a look at Fig. 5.18. This offers a kind of menu choice of headings, but it's done by using RESTORE followed by a line number. When you pick a number, it is used in line 40 to carry out a RESTORE command which has a line number following it. RESTORE 2000, for example, means start reading DATA at line 2000. Each DATA line contains four items, so that when line

```
10 PRINT"Which list do you want?"
20 PRINT:INPUT"Number 1 to 3, please
";A%
30 IF A%<1 OR A%>3 THEN PRINT"Between
 1 and 3 only, please":GOTO 10
40 IF A%=1 THEN RESTORE 1000 ELSE IF
A%=2 THEN RESTORE 2000 ELSE RESTORE 3
000
50 FOR N%=1 TO 4:READ A$:PRINTA$:NEXT
60 END
1000 DATA Austin, Rover, Triumph,Jagu
ar
2000 DATA BMW, Porsche,Mercedes, Opel
3000 DATA Alfa Romeo, Lancia, Fiat, F
errari
```

Fig. 5.18. How RESTORE can be used to select different DATA lines.

50 is carried out, four items will be read from whichever line has been picked. It's a useful way of selecting from a number of lists which will be used each time the program is used.

Loops and arrays

The loop commands of any computer are particularly useful when you have to deal with array variables. The reason is that you can set up a loop, such as a FOR N%=1 TO 100 loop, and make the array items use the counter variable N%, as, for example, A%(N%). This can make the actions of filling or printing an array look very simple pieces of programming. Figure 5.19

```
10 CLS
20 DIM A%(20):FOR N%=1 TO 20
30 A%(N%)=RND(1)*100+1
40 NEXT
50 PRINT
60 PRINTTAB(13)"Marks List"
70 PRINT:FOR N%=1 TO 20
80 PRINT"Item";N%;" received";A%(N%);
" marks."
90 NEXT
```

Fig. 5.19. An array of subscripted number variables being assigned in a loop.

illustrates this. Lines 10 to 40 generate an (imaginary) set of twenty examination marks. This is done simply to avoid the hard work of entering the real thing! Line 20 dimensions the integer number array A% to a maximum of twenty items. If we had needed only up to ten items, we could have dispensed with this DIM line, but it's always better to include it *even for small arrays*, just to remind yourself that you're dealing with an array. The array variable in line 30 is a *subscripted number variable*, and the *subscript* is the number that is represented by N%. Each item is obtained by finding a random number between 1 and 100, and is then assigned to A%(N%). Twenty of these 'marks' are assigned in this way, and then lines 60 to 90 print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

Figure 5.20 extends this another step further. This time you are invited to type a name and a mark for each of ten items. When the list is complete, the screen is cleared and a total variable is set to zero in line 70. The list is then

```
10 CLS:PRINT:CLEAR 500
20 PRINT"Please enter names and marks
."
30 DIM N$(10),A%(10):FOR N%=1 TO 10
40 PRINT"Name  - ";:INPUT N$(N%)
50 PRINT"Mark  - ";:INPUT A%(N%)
60 NEXT
70 CLS:T%=0
100 PRINTTAB(13)"MARKS LIST":PRINT
110 FOR N%=1 TO 10
120 PRINTTAB(2)N$(N%);TAB(22)A%(N%)
130 T%=T%+A%(N%)
140 NEXT
150 PRINT
160 PRINT"AVERAGE IS";T%/(N%-1)
```

Fig. 5.20. Using strings in one array and numbers in another.

printed neatly, and on each pass through the loop the total is counted up (in line 130) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this array form. This example uses both a string array (names) and a number array (marks). Remember that in any program like this, the arrays will have to be dimensioned correctly. If you don't know what number to expect when you write the program, you will have to add a line early in the program which reads, for example:

INPUT"How many items ";A%:DIM N%(A%),NM$(A%)

so that the user has to specify how many items there will be in the array. The only way that an array can be created without dimensioning it is when the array is created on tape or on disk, and that's something quite different which we shall look at later in Chapter 11.

Rows and columns

You can imagine an array as a list of items, one after the other, but there is a variety of array which allows a different kind of list, called a *matrix*. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Fig. 5.21 to see how this works. We use here a variable N$ which has two

```
10 CLS
20 FOR N%=1 TO 3
30 FOR J%=1 TO 2
40 READ N$(N%,J%)
50 NEXT J%,N%
60 FOR N%=1 TO 3
70 PRINTTAB(5)N$(N%,1);TAB(25)N$(N%,2
)
80 NEXT
100 DATA Horse,Foal,Cow,Calf,Dog,Pupp
y
```

Fig. 5.21. Making a matrix of rows and columns.

subscript numbers. The first number is the row number, the second is the column number, and we need two FOR...NEXT loops to read data into this matrix. This is carried out in lines 20 to 50. Notice the shortened NEXT J%,N% in line 50, which is a way of writing NEXT J%:NEXT N%. The items are then printed in columns by the loop in lines 60 to 80. In this loop, the variable N% is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively.

Figure 5.22 shows a much more ambitious matrix program. This one uses a row number for matrix A$ which is 50, and so it has to be dimensioned in line 10. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 20 to 60. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out tests of inputs (mugtraps) just to keep this example reasonably short, but you would certainly need some sort of mugtraps, even if only to avoid things like entering two letters or a whole name at line 100. The choice here is the entry of a first (capital) letter, and we should really check that this *is* a capital letter. If a lower-case letter is entered, it can easily be tested for because its ASCII code will be more than 96. We can convert a lower-case letter into an upper-case letter if we subtract 32 from the ASCII code. A step like:

```
J$=CHR$(ASC(J$)−32)
```

will carry out the conversion.

```
10 CLS:DIM A$(50,2)
20 FOR N%=1 TO 50
30 PRINT"Name ";:INPUT A$(N%,1)
40 PRINT"Tel. No. ";:INPUT A$(N%,2)
50 PRINT:PRINT
60 NEXT
70 CLS:PRINT
80 PRINT"List Complete"
90 PRINT:PRINT"Pick an initial letter
...":PRINT"Use X to end program.":Q=0
100 INPUT J$:IF J$="X" THEN 160
110 FOR N%=1 TO 50
120 IF J$=LEFT$(A$(N%,1),1)THEN PRINT
"Name is ";A$(N%,1):PRINT"Number is "
;A$(N%,2):Q=1
130 NEXT
140 IF Q=0THEN PRINT"Not found...":PR
INT
150 GOTO 90
160 PRINT"End of program"
```

Fig. 5.22. Using a name and number matrix for a simple telephone directory application.

The next part of the program deals with picking a name by specifying an initial letter. The important point here is that if we specify J, for example, it should not just pick out the *first* name that starts with J. That way you always get Jim, and never get John! In addition, if there is no name in the list which starts with the letter that you want, you should be told about this. You should also be told how to leave the loop, because this is a GOTO type of loop. Line 100 deals with the input, and a choice of X here will end the program. If any other letter has been selected, a loop starts in line 110. Each name is selected in turn by the loop, and the first letter of the name is compared with the letter which was selected. If the two match, then the whole name and telephone number will be printed. At the same time, a variable Q (shouldn't it have been Q%?) is set to 1. This variable was made equal to 0 before the loop started, and it is used as a signal that a name has been found. The NEXT in line 130 marks the limit of this loop.

When line 140 runs, Q will be zero if no names have been found. The message will then be printed. If a name *has* been found then Q will be 1, and the message is not printed. In either case, the GOTO 90 in line 150 forces this selection part of the program to repeat until you type X in response to the INPUT step.

The next thing that you might need to do with sets of names and numbers of this type is to record them. You can do this easily with the cassette recorder, and that's a subject that we shall look at in the course of Chapter 11. If you have a disk system, however, you automatically have a set of extra commands which will open up a whole new world of data processing to you.

Chapter Six
Menus and Subroutines

We have seen how RESTORE can be used to make a choice of items that are to be read from a list. Very often, though, we want to present a user with a menu on the screen. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. We could use a set of lines such as:

```
IF K =1 THEN 1000
IF K =2 THEN 2000
```

and so on. There is a much simpler method, however, which uses a new instruction ON N% GOTO, where N% is a number variable, an integer in this example. You can use any number variable, of course, not just N%.

Figure 6.1 shows a typical menu that uses this instruction. Line 10 removes the KEY display, and clears the screen. Lines 20 to 80 then present the menu items on the screen, and line 100 invites you to pick one item by typing its number. The INKEY$ loop in line 110 keeps the program looking for a key until you make your choice, and then line 120 tests your choice with a mugtrap. VAL has to be used, remember, because INKEY$ produces a string variable, and you can't compare a string with a number (nor a rose with a carrot). By using K%=VAL(K$) you get an integer number variable K% which will hold a number that is in the correct form to be compared. If you had pressed a letter key then K% would be zero.

The choice is then made in line 130, with the ON K% GOTO instruction. Now what happens here? If K% equals 1, then the first line number that follows GOTO is used. If K% equals 2, then the second line number following GOTO is used, and so on. All that you have to do is to arrange the line numbers in the same order as your choices. You needn't have a list that looks neat. A line such as ON K% GOTO 50,216,484,714,1000 would be just as satisfactory so long as these numbers contained the start of routines that dealt with the menu choices. In this example, the line numbers simply lead to PRINT instructions so as to keep the example reasonably short. Note that the last item in a menu like this should always be a QUIT option, meaning

```
10 KEY OFF:CLS
20 PRINTTAB(16)"MENU"
30 PRINT:PRINT
40 PRINT"1. Enter names."
50 PRINT"2. Enter phone numbers."
60 PRINT"3. List all names."
70 PRINT"4. List local numbers."
80 PRINT"5. End program."
90 PRINT
100 PRINT"Please select by number 1 t
o 5"
110 K$=INKEY$:IF K$=""THEN 110
120 K%=VAL(K$):IF K%<1 OR K%>5 THEN P
RINT"Incorrect choice- please try aga
in":GOTO 100
130 ON K% GOTO 1000,2000,3000,4000,50
00
140 KEY ON:END
150 PRINT"Names here":GOTO 140
1000 PRINT"Names here":GOTO 140
2000 PRINT"Numbers here":GOTO 140
3000 PRINT"List of names.":GOTO 140
4000 PRINT"Local numbers here":GOTO 1
40
5000 PRINT"END":GOTO 140
```

Fig. 6.1. A menu choice which uses the ON K% GOTO instruction.

one that lets you leave the program. There is nothing quite so frustrating as a program that won't let you get away!

This type of menu selection is useful, but an even more useful method makes use of *subroutines*. A subroutine is a section of program which can be inserted anywhere that you like in a longer program. A subroutine is inserted by typing the instruction word GOSUB, followed by the line number in which the subroutine starts. When your program comes to this instruction, it will jump to the line number that follows GOSUB, just as if you had used GOTO. Unlike GOTO, however, GOSUB offers an *automatic* return. The word RETURN is used at the end of the subroutine lines, and it will cause the program to return to the point immediately following the GOSUB. Figure 6.2 illustrates this. When the program runs, line 20 assigns a phrase to the string variable T$. The next line is GOSUB 1000, which means that the program must jump to the routine which starts at line 1000. In this line L%, the number of characters in T$, is found. The following line 1010 then prints T$ centred on the screen. Line 1020 consists of the word RETURN. As the name suggests, this means that the program must return to a position that is immediately following the GOSUB. In this first case, that means to line 40. This carries out another assignment of T$, this time to a string of underline dashes. Once again, calling GOSUB 1000 in line 50 will

```
10 CLS
20 T$="MSX Computing"
30 GOSUB 1000
40 T$=STRING$(LEN(T$),"_")
50 GOSUB 1000
60 LOCATE 2,4
70 PRINT"Neat, isn't it?"
80 END
1000 L%=LEN(T$)
1010 PRINTTAB((37-L%)/2);T$
1020 RETURN
```

Fig. 6.2. Using a subroutine – this is the key to more advanced programming.

cause this new value of T$ to be printed centred, and the RETURN this time makes the program return to line 60. With a GOTO, you are stuck with just one destination line number, but the RETURN at the end of a GOSUB makes sure you return to the command which follows the GOSUB. Even if you have a multistatement line like:

T$="MENU":GOSUB 1000:PRINT"NOTES"

then the subroutine will return correctly, in this case to perform the PRINT action.

Now for its application to menus, Fig. 6.3 shows subroutines in use as part of a (totally imaginary) games program. Lines 10 to 80 offer a choice, and line 90 invites you to choose. The familiar INKEY$ and mugtrap actions follow, and then line 120 causes the choice to be carried out. This time, however, the program will return to whatever follows the choice. For example, if you pressed key 1, then the subroutine that starts at line 1000 is carried out, and the program returns to line 120 to check if you might also want subroutines 2000, 3000, 4000 or 5000. Since the value of K% is still 1, the program then goes to line 130 and ends. If line 1000 had altered the value of K%, however, you could find that a second subroutine was selected following the first one. Never make any other use of the variable name that you have selected for ON K% GOSUB.

A subroutine is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. Take a look at Fig. 6.4 by way of an example. The subroutine is an elaboration on the INKEY$ routine. The trouble with INKEY$ is that it doesn't remind you that it's in use; there's no question mark printed as there is when you use INPUT. The subroutine in lines 1000 to 1040 remedies that by causing an asterisk to flash while you are thinking about which key to press. The asterisk is flashed by alternately printing the asterisk and some delete step. According to some of the MSX manuals, CHR$(8) should make the cursor backspace and delete the character under it. On the machine which I used, the cursor backspaced but did not delete. The line 1030

```
10 CLS:PRINT
20 PRINTTAB(8)"Choose your monster."
30 PRINT
40 PRINTTAB(2)"1. Vampire."
50 PRINTTAB(2)"2. Werewolf."
60 PRINTTAB(2)"3. Zombie."
70 PRINTTAB(2)"4. Sgt. Major."
80 PRINTTAB(2)"5. Flying picket."
90 PRINT:PRINT"Select by number, plea
se":PRINT:PRINT
100 GOSUB 10000:REM INKEY$ ROUTINE
110 IF K%<1 OR K%>5 THEN PRINT"Faulty
 selection- 1 to 5 only-":PRINT"Pleas
e try again.":GOTO100
120 ON K% GOSUB 1000,2000,3000,4000,5
000
130 PRINT:PRINT"Want another choice?
Type y or n"
140 GOSUB 10000:IF K$="y" OR K$="Y" T
HEN 10
150 END
1000 PRINT"Blood, blood, bootiful blo
od":RETURN
2000 PRINT"Howl, snarl, gnash":RETURN
3000 PRINT"I obey, master, I obey":RE
TURN
4000 PRINT"You 'orrible little man":R
ETURN
5000 PRINT"Blood, howl, I obey, smash"
:RETURN
10000 K$=INKEY$:IF K$=""THEN 10000 EL
SE K%=VAL(K$)
10010 RETURN
```

Fig. 6.3. A menu choice for an imaginary game that makes use of subroutines.

therefore uses CHR$(8) to backspace, CHR$(32) to print a space, and CHR$(29) to backspace again. On this machine, I found that CHR$(8) and CHR$(29) had exactly the same effect. To make the rate of flashing reasonably slow, I've added another subroutine, a delay in line 2000.

While we're on the subject of menus, there's another subroutine, in Fig. 6.5, which can make a menu look a lot more interesting. This is a visual menu choice, and its use brings several advantages to your menus. One is that you don't need to have the items of the menu numbered, because you don't choose by number. Instead, a little arrow flashes next to the first item of the menu. This arrow can be shifted by using the cursor keys, the ones which are marked with the vertical up or down arrows. Since the program makes it impossible to shift the arrow beyond the menu items, no sort of testing or mugtrapping of the answer is needed. The choice is passed back to

```
10 CLS
20 PRINT"Choose 1 or 2,please"
30 GOSUB 1000
40 PRINT"Your choice was ";K$
50 END
1000 K$=INKEY$
1010 IF K$<>""THEN RETURN
1020 PRINT"*";:GOSUB2000
1030 PRINT CHR$(8);CHR$(32);CHR$(29);
:GOSUB 2000
1040 GOTO 1000
2000 FOR J=1 TO 200:NEXT:RETURN
```

Fig. 6.4. A flashing asterisk subroutine. The asterisk flashes until you press a key.

```
10 CLS:KEY OFF
20 T$="Your Choice"
30 ST%=2:NR%=4
40 GOSUB 10000
50 LOCATE2,12:PRINT"You chose option
";CH%
60 KEY ON:END
10000 PRINTTAB((37-LEN(T$))/2);T$
10010 FOR J%=1 TO NR%
10020 LOCATE 3,ST%+J%-1
10030 READ MENU$:PRINT MENU$
10040 NEXT:PS%=ST%
10050 LOCATE 1,PS%:PRINTCHR$(175)
10060 FOR J%=1 TO 200:NEXT
10070 LOCATE 1,PS%:PRINTCHR$(32)
10080 FOR J%=1 TO 200:NEXT
10090 K$=INKEY$
10100 IF K$=CHR$(32)THEN CH%=PS%-ST%+
1:RETURN
10110 IF K$=CHR$(30)THEN PS%=PS%-1
10120 IF K$=CHR$(31)THEN PS%=PS%+1
10130 IF PS%>ST%+NR%-1 THEN PS%=ST%
10140 IF PS%<ST%THEN PS%=ST%+NR%-1
10150 GOTO 10050
10160 DATA Input Data,Output Data,Che
ck Data,Alter Data
```

Fig. 6.5. A visual menu subroutine. You use the cursor keys to move the arrow, then press the spacebar when the arrow points to the item that you want. The subroutine has been written so that you can easily use this in your own programs.

the main routine as a number CH%, which you can then use in a line such as: ON CH% GOSUB 1000,2000,3000,4000 and so on. Try it for yourself, and see how much better it looks as compared to the traditional menu.

The subroutine needs to have some values passed to it. The title is passed as T$, and two integer numbers are needed also. One of these is ST%, which is the line at which the first item of the menu will appear. The other is NR%, which is the number of items on the menu. The actual menu items are placed in a DATA line which can be anywhere in the program. If you have more than one menu, you can use RESTORE to get the correct set of data items. Once these quantities have been assigned, the subroutine can be called. In the example, the numbers have been set up to start on line 2 and use four items only.

The subroutine starts in line 10000 to 10040 by printing the title, centred, and then reading the menu items and printing them. Variable ST% is used to make sure that the items are placed on the correct lines. The LOCATE command makes sure that the items are all tabbed to column number 3 (the fourth column, since counting starts at 0, remember). At the end of line 10040, using PS%=ST% passes the value of ST% (four in this example) to another variable PS% which will be used to control the position of the arrow-head. Line 10050 then starts a loop which will print the arrow-head, wait, delete the arrow, wait, and then look for a key being pressed. If this key is the spacebar, then the program assigns CH% and returns. If the key is a cursor key, the arrow-head is moved. The movement is then checked to make sure that it cannot be above or below the menu items.

Line 10050 to 10080 print the arrow-head, wait, print a space, and wait again. The pause could have been put into another subroutine, and if you have a pause subroutine in your program anyway you would use it in place of lines 10060 and 10080. Line 10090 is the INKEY$ line – note that we *don't* use IF K$="" THEN 10090 here, because we do not want the program to hang up at this point. If the program hangs up, then the arrow-head doesn't flash! The next three lines 100100 to 10120, test K$. If this was the spacebar (ASCII code 32) then the value of CH% is obtained from PS%–ST%+1. The idea is that ST% is the number of the first screen line which contains a menu item, and PS% is the one that the arrow points to. If the arrow is still on the first line, PS%–ST%+1 is 1–1+1=1; if the arrow is on the second line, then PS%–ST%+1 is 2–1+1=2 and so on. If you only use the menu subroutine once, then you can substitute numbers in place of ST% and NR%. Moving on, lines 10110 and 10120 test for the cursor keys, and alter the value of PS% accordingly. Lines 10130 and 10140 then test the value of PS%. If this has gone out of limits, then it is returned to the opposite limit. If PS% would place the arrow above the top menu item, it's placed instead at the bottom item. If the value of PS% is such that it would put the arrow below the bottom item, then it is returned to the top. This sort of action is called *wraparound*. Finally, line 10150 is the GOTO which completes the loop. The loop is broken only when the spacebar is pressed. You could, of course, alter this so that the ESC or TAB or any other key operated this action. Now try it out in your own programs!

Rolling your own

You can get a lot of enjoyment from your MSX computer when you use it to enter programs from cassettes that you have bought, or from plug-in cartridges. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that. After all, buying a computer and not programming it yourself is like buying a BMW and getting someone else to drive it for you.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called *database* programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move across the screen. Programs of that type need instructions that we shall be looking at in detail in the next few chapters. What we are going to look at in this section is how a program can be designed using *subroutines* because this is a design method that can be used for all types of programs. Once you can design simple programs of this type you can progress, using the same methods, to design your own graphics and sound programs. Remember, though, that most of the very fast moving or elaborate graphics programs that you see are not written in BASIC. The reason is that BASIC is too slow to allow fast movement, or the control of lots of moving objects. These arcade-type programs that you can buy are written in *machine code*, a set of number-coded instructions direct to the microprocessor that is the heart of the computer. This bypasses BASIC altogether, and is very much more difficult. If you learn how to design programs in BASIC, however, you will be able to learn machine code later. All you need is experience – a lot of it.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. We have already dabbled with the design of simple programs, and I want to show you that this is *all you ever need!* The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. I use a student's pad of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program, but it will illustrate all the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly *structured* program. Structured in this sense means that the program is put together in a way that is a logical sequence, so that it is easy to add to, change, or redesign. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you intended at first.

As an example, take a look at Fig. 6.6. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the names of animals and their young. The program plan shows what I expect of this game. It must present the name of an animal, picked at random, on the screen, and then ask what the name of its young is. A little bit more thought produces some additional points. The name of the young animal will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question.

Aims
1. Present the name of an animal on the screen.
2. Ask what its young is called.
3. Reply must be correctly spelled.
4. User must not be able to read the answer from a listing.
5. Give one point for each correct answer.
6. Allow two chances at each question.
7. Keep a track of the number of attempts.
8. Present the score as the number of successes out of the number of attempts.
9. Pick animal names at random.

Fig. 6.6. A program outline plan. This is your starter!

Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 6.7 shows what this might look like at this stage.

Foundation stones

Now, at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.8 shows what you should aim for at this stage. There are only fourteen lines of program here, and that's as much as you want. This is a foundation, remember, not the Empire State Building! It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM. REM means REMinder, and any line of a program that starts with REM will be ignored

```
Title

Instructions

Repeat –
   Name of animal                 { Pick random number
                                  { Use to select from array

   Ask for name of young          { Use INPUT
                                  { Select ASCII codes from array
   Compare with correct name      { Decode to answer
                                  { Compare
                                  { +1 if correct
   Score                          { Try again if not
                                  { Abandon after second attempt

   Ask if another wanted          { Y or N

Until answer not 'Y' or 'y'

End
```

Fig. 6.7. The next stage in expanding the outline.

by the computer. This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you LIST. In Fig. 6.8, I have put the REM notes on lines which are numbered just 1 more than the main lines.

```
10 CLS:GOSUB 1000
11 REM Title
20 GOSUB 1200
21 REM Instructions
30 GOSUB 1400
31 REM Setup
40 GOSUB 2000
41 REM Play
50 GOSUB 3000
51 REM Score
60 GOSUB 4000
61 REM Another?
70 IF INSTR("YESyes",K$)<>0 THEN 40
80 END
```

Fig. 6.8. A *core* or *foundation* program for the example.

This way, I can remove all the REM lines later. How much later? When the program is complete, tested, and working perfectly. REMs are useful, but they make a program take up more space in memory, and run slightly slower. I always like to keep one copy of a program with the REMs in place, and another 'working' copy which has no REMs. That way I have a fast and efficient program for everyday use, and a fully-detailed version that I can use if I want to make changes.

Let's get back to the program itself. As you can see, it consists of a set of GOSUB instructions, with references to lines that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 6.7 exactly, and the only part that is not committed to a GOSUB is the IF in line 70. What we shall do is to write a subroutine which will use INKEY$ to look for a y or Y being pressed, and line 70 deals with the answer. What's the question? Why, it's the Do you want another game step that we planned for earlier.

Line 70 makes use of that INSTR keyword which we looked at earlier. By testing with INSTR("YESyes",K$), we will get 1 if Y is pressed and 4 if y is pressed. If K$ is neither y nor Y, then INSTR gives 0, meaning that the string we are seeking is not contained in YESyes. Simple, but very useful. We could have used INSTR("Yy",K$) in this example, since only one letter is being tested. I have used the full form here, because it makes the subroutine more generally useful.

Take a good long look at this fourteen-line piece of program, because it's important. The use of all the subroutines means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the subroutines. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
9 GOTO 30
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it.

The next step is to get to the keyboard (at last, at last!) and enter this core program. If you use the GOTO step to skip round the title and instructions temporarily, you can then put in simple PRINT lines at each subroutine line number. We did this, you remember, in the program of Fig. 6.1, so you know how to go about it. This allows you to test your core program and be sure that it will work before you go any further.

The next step is to record this core program and then keep adding to the core. If you have the core recorded, then you can load this into your computer, add one of the subroutines, and then test. When you are satisfied that it works, you can record the whole lot on another cassette. Next time you want to add a subroutine, you start with this version, and so on. This way, you keep tapes of a steadily growing program, with each stage tested and known to work. Again, this is important. Very often, testing takes longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each subroutine as you go, you know that you can have confidence in the earlier parts of the program, and you can concentrate on errors in the new sections.

Subroutine routine

The next thing we have to do is to design the subroutines. Now some of these may not need much designing. Take, for example, the subroutine that is to be placed in line 4000. This is just our familiar INKEY$ routine, along with a bit of PRINT, so we can deal with it right away. Figure 6.9 shows the form it might take. The subroutine is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this subroutine in place.

```
4000 PRINT"Would you like another one
?"
4010 PRINT"Please answer y or n."
4020 K$=INKEY$:IF K$=""THEN 4020
4030 RETURN
```

Fig. 6.9. The INKEY$ subroutine for line 4000.

Now we come to what you might think is the hardest part of the job – the subroutine which carries out the Play action. In fact, you don't have to learn anything new to do this. The Play subroutine is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a subroutine to be dealt with later.

As an example, take a look at Fig. 6.10. This is a plan for the Play subroutine, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of a string of ASCII codes, with each code written as a three-figure number. Why three figures? Well, the capital letters will use two figures only, the small letters three, so making them all into three figures simplifies things.

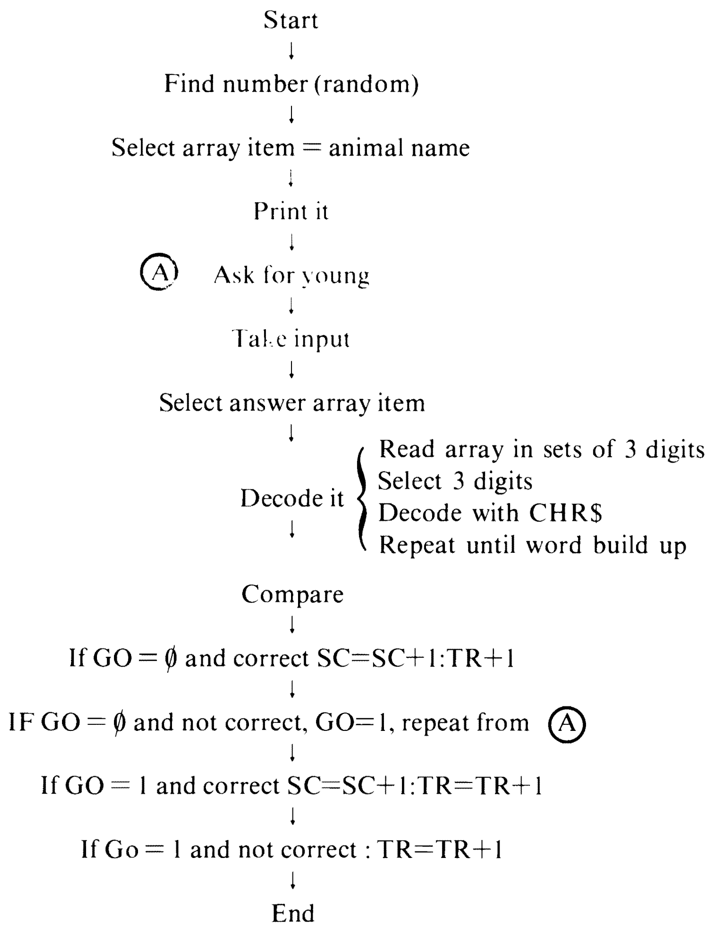

Start
Find number (random)
Select array item = animal name
Print it
A
Ask for young
Take input
Select answer array item
Decode it
Read array in sets of 3 digits
Select 3 digits
Decode with CHR$
Repeat until word build up
Compare
If GO = ∅ and correct SC=SC+1:TR+1
IF GO = ∅ and not correct, GO=1, repeat from A
If GO = 1 and correct SC=SC+1:TR=TR+1
If Go = 1 and not correct : TR=TR+1
End

Fig. 6.10. Planning the Play subroutine.

You'll see why later – what we do is to write a number like 86 as 086, and so on. That's the first item for this subroutine.

The next one is that we shall keep the names of the animals in an array. This has several advantages. One is that it's beautifully easy to select the name of an animal at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 10, then we can place the answers in DATA lines, one set of numbers in each data line, and read these also as a string array. The alternative would be to keep the names and the answers in DATA lines, and use RESTORE. This is not quite so neat, however.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using SC% for the score and TR% for the number of tries look self-explanatory. The third one, GO% is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the animal names – Q$.

Play for today

Figure 6.11 shows what I've ended up with as a result of the plan in Fig. 6.10. The steps are to pick a random number, use it to print an animal name, and

```
2000 GO%=0:V%=INT(10*RND(1))+1
2010 CLS:PRINT"The animal is - ";Q$(V
%)
2020 PRINT:PRINT"The young is called
- ";
2030 INPUT X$:TR%=TR%+1
2040 GOSUB 5000
2041 REM Find correct answer
2050 RETURN
```

Fig. 6.11. The program lines for the Play subroutine.

then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another subroutine. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time. As it is, I've had to put another subroutine into this one to keep things short.

We start the subroutine at line 2000 by 'clearing a variable'. The size of GO% is set to 0, to make sure that this variable has the correct size each time this subroutine is started. The second part of line 2000 then picks a number, at random, lying between 1 and 10. Lines 2010 to 2030 are straightforward stuff. We print the name of the animal that corresponds to the random number, and ask for an answer, the young of that animal. The last section of line 2030 counts the number of attempts. This is the logical place to put this step, because we want to make the count each time there is an answer. Now it's chicken-out time. I don't want to get involved in the reading of ASCII codes right now, so I'll leave it to a subroutine, starting in line 5000, which I'll write later. The REM in line 2041 reminds me what this new subroutine will have to do, and the Play subroutine ends with the usual RETURN.

Down among the details

With the Play subroutine safely on tape, we can think now about the details. The first one to look at should be one that precedes or follows the Play step, and I've chosen the Score routine. As usual, it has to be planned, and Fig. 6.12 shows the plan. Each time there is a correct answer, the number variable SC% will be incremented, and we can go back to the main program.

Answer correct	Increment SC Increment TR GO=0	Next question
Answer incorrect GO=0	Make GO=1 Get another answer	
Answer incorrect GO=1	Make GO=0 Move to next question	

Fig. 6.12. Planning the Score subroutine.

More is needed if the answer does not match exactly. We need to print a message, and allow another go. If the result of this next go is not correct, that's an end to the attempts. At this point, you might later want to include some sound. We could have a short beep to announce a mistake, and a long one for a correct answer. Write it down!

Figure 6.13 shows the program subroutine that has been developed from this plan. Line 3000 deals with a correct answer. Since we need to print a message which would not fit in a line, we use GOTO 3200 to finish the job. The GOTO 3040 in line 3210 ensures that if the answer was correct, the rest of the subroutine is skipped, and the subroutine returns. If the answer is not correct, though, line 3010 swings into action. This tests the value of GO% and if it is zero causes a jump to line 3300 to print its message and give

```
3000 PRINT:IF X$=A$ THEN SC%=SC%+1:GO
TO 3200
3010 IF GO%=0 THEN GOTO 3300
3020 GO%=0:PRINT"No luck - try the ne
xt one."
3030 FOR Q=1 TO 1000:NEXT
3040 RETURN
3200 PRINT"Correct- your score is now
 ";SC%
3210 PRINT"in ";TR%;" attempts. ":GOS
UB 7000:GOTO 3040
3300 PRINT"Not correct- but it might
be your"
3310 PRINT"spelling! You get another
go free.":TR%=TR%-1
3320 GOSUB 7000:GO%=1:GOSUB 2010:GOTO
 3000
```

Fig. 6.13. The Score subroutine written.

further instructions. Line 3320 calls the subroutine at line 2010 again so that the user can make another answer entry. The GOTO 3000 at the end of line 3320 then tests this answer again.

Now there's a piece of cunning here. The number variable GO% starts with a value of 0. When there is a correct answer, however, and GO% is still 0, line 3010 is carried out. One of the actions of line 3320, however, is to set GO% to 1. When you answer again, with GO%=1, line 3000 will be used, and if your second answer is wrong, line 3010 cannot be used, because GO% is not zero. The next line that is tried, then, is 3020. This puts GO% back to zero for the next round, prints a sympathetic message, pauses, and then lets the subroutine return in line 3040.

Now that we've got the bit between our teeth, we can polish off the rest of the subroutines. Figure 6.14 shows the subroutine that deals with

```
1400 TR%=0:SC%=0:GO%=0:V%=RND(-TIME)
1410 DIM Q$(10),A$(10)
1420 FOR J%=1 TO 10:READ Q$(J%):NEXT
1430 FOR J%=1 TO 10:READ A$(J%):NEXT
1440 RETURN
```

Fig. 6.14. The dimensioning and array subroutine.

dimensioning and arrays. Line 1400 sets all the variables for the scoring system to zero, and makes sure that the same sequence is not repeated each time you use the program. Line 1410 dimensions the array Q$ that will be used for the names of the animals, and A$ which will be used for the numbers that give the answers. Line 1420 then reads the names from a data list into the array Q$, and line 1430 reads the numbers into A$ – and that's it! We can write the DATA lines later, as usual.

Next comes the business of finding the answer. We have planned this, so it shouldn't need too much hassle. Figure 6.15 shows the program lines. The variable V% is the one that we have selected at random, and it's used to select one of the strings of ASCII numbers, A$(V%). Since each number consists of three digits, we want to slice this string three digits at a time, and that's why we use STEP 3 in the FOR...NEXT loop in line 5000. Line 5010 then builds up the answer string, which we call A$. Remember that A$, used alone, is not confused with the A$(V%) array. A$ is set to a blank in the first part of line 5000 to ensure that we always start with a blank string, not with

```
5000 A$="":FOR J%=1 TO LEN(A$(V%))STE
P3
5010 A$=A$+CHR$(VAL(MID$(A$(V%),J%,3)
)):NEXT
5020 RETURN
```

Fig. 6.15. Checking the answer.

the previous answer, which would also be A$. The string A$ is then built up by selecting three digits, converting to the form of a number by using VAL, then to a character by using CHR$. Remember that when you have a lot of brackets like this, you read from the innermost set to the outermost. This character is then added to A$, and then continues until all the numbers in the string have been dealt with. That's the hard work over. Figure 6.16 is the

```
1200 CLS:PRINTTAB(12)"INSTRUCTIONS"
1210 PRINT:PRINTTAB(2)"The computer w
ill supply you with"
1220 PRINT"the name of an animal. You
 should "
1230 PRINT"type the name of its young
 - and "
1240 PRINT"make sure that your spelli
ng is "
1250 PRINT "correct, and that you sta
rt each "
1260 PRINT "name with a capital lette
r. The"
1270 PRINT"computer will keep score f
or you."
1280 PRINT"You get two shots at each
name."
1290 PRINT:PRINT"Press the spacebar t
o start."
1300 IF INKEY$<>" " THEN 1300 ELSE RE
TURN
```

Fig. 6.16. The instructions. Always leave these until almost finished.

subroutine for the instructions, and Fig. 6.17 is the title subroutine. The title lines include a pause, and have been written with a SCREEN 1 type of display. We'll deal with this in more detail in Chapter 7 – it gives slightly larger letters which are more suited to a heading. Finally, Fig. 6.18 shows the DATA lines.

Now we can put it all together and try it out. Because it's been designed in sections like this, it's easy for you to modify it. I have chosen a very simple theme just for this purpose. You can use different DATA, for example. You can use a lot more data – but remember to change the DIM in line 1410. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. You can add some sound

```
1000 SCREEN 1
1010 PRINTTAB(8)"Young Animals"
1020 GOSUB 7000:SCREEN 0:RETURN
```

Fig. 6.17. The title program lines.

```
6000 DATA Dog,Cat,Cow,Horse,Hen,Fox,K
angaroo,Goose,Lion,Pig
6001 DATA 080117112112121
6002 DATA 075105116116101110
6003 DATA 067097108102
6004 DATA 070111097108
6005 DATA 067104105099107101110
6006 DATA 067117098
6007 DATA 074111101121
6008 DATA 071111115108105110103
6009 DATA 067117098
6010 DATA080105103108101116
7000 FOR Q=1 TO 3000:NEXT:RETURN
```

Fig. 6.18. The DATA lines that are needed, along with a time delay subroutine.

effects, for example, or add more interesting graphics. One major fault of the program is that once an item has been used, it can be picked again, because that's the sort of thing that RND can cause. You can get round this by swapping the item that has been picked with the last item (unless it was the last item), and then cutting down the number that you can pick from. For example, if you picked number 5, swap numbers 5 and 10, then pick from 9. This means that the 10*RND(1)+1 step will become D%*RND (1)+1, where D% starts at 10, and is reduced by 1 each time a question has been answered correctly.

There's a lot, in fact, that you can do to make this program into something much more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to rebuild any way you like. It will give you some idea of the sense of achievement that you can get from mastering your MSX computer. As your experience grows, you will then be able to design programs that are very much longer and more elaborate than this one by a long way. By that time, you'll be thinking of adding a printer and a disk drive to your MSX computer. Go ahead; they will open up a whole new world of MSX computing to you.

Chapter Seven
Special Effects and Geometrical Shapes

Any modern computer is expected to be able to produce dazzling displays of colour and other special effects. The MSX computer is no exception, and in this chapter, we'll start to look at some of the effects that are possible. To start with, we have to know some of the terms that are used, and the first of these is *graphics*. Graphics means pictures that can be drawn on the screen, and all modern computers have instructions that allow you to draw such patterns. In connection with these patterns, you'll see the words low resolution and high resolution used. Resolution isn't such an easy term to explain. Imagine that you are creating pictures on a paper sheet about eleven inches across by eight inches deep. That's roughly the size of a TV screen that is described as being a 14 inch screen (it's about 14 inches diagonally!).

Now if you are asked to create the pictures by using rectangles of coloured paper, you are dealing with picture-making in a way that is very similar to the way that the computer operates. Suppose that you are allowed only 888 pieces of paper, of such a size that all 888 put into place will fill the screen. You couldn't draw very finely detailed pictures with this comparatively small number of large pieces, and this is what we mean by low resolution. On the other hand, if you were provided with pieces so small that you would need 49152 of them to fill an entire screen size, you could produce very much more detailed pictures. This is what we mean by high resolution. The MSX computer has both low and high resolution graphics available, and the figures that I have used correspond to the size of the blocks that the MSX computer uses. In this chapter, we're going to deal with the low resolution graphics, and some of the commands for the high resolution graphics of the MSX computer. There are three points in particular that we have to look at. These are how to obtain graphics characters, how to place them on the screen, and how to make shapes of our own design.

Keyboard graphics

The graphics shapes that are illustrated in the MSX computer manuals can all be obtained by pressing keys on the keyboard. The difference is that you have to press the graphics key, labelled GRAPH, as well. You can obtain another set of graphics characters if you press the SHIFT key in addition to the GRAPHICS key and a letter/number key. These graphics characters can be printed in the same way as you print words, by using the PRINT command, followed by a quote, then typing the graphics characters, then ending with another quotemark. If you want to use the characters to make fancy underlining, or to provide shapes to identify menu choices, this is one way to do it. Unfortunately, I can't illustrate this in a program, because printers generally won't display these shapes as they appear on the screen.

The character codes

The alternative method, which allows us a lot more scope for illustration is to use the ASCII codes for the characters. These are shown in the MSX computer manuals as well, but it's not very easy to see in some manuals what numbers give you the graphics shapes. The program in Fig. 7.1 will remind

```
10 CLS:FOR N%=192 TO 223 STEP 16
20 FOR J%=0 TO 15
30 PRINTCHR$(J%+N%);" ";:NEXT
40 PRINT: PRINT:NEXT
50 FOR N%=64 TO 95 STEP 16
60 FOR J%=0 TO 15
70 PRINTCHR$(1);CHR$(J%+N%);" ";:NEXT
80 PRINT:PRINT:NEXT
```

Fig. 7.1. A program which prints the graphics shapes on to the screen.

you of them. Only certain code numbers are used for graphics, and there are two sets. One set uses ASCII code numbers 192 to 223 (32 characters altogether), and the other set uses ASCII codes 64 to 95, another 32 characters. These codes, 64 to 95 are normally used for letters, and to get the graphics shapes you have to precede each code with CHR$(1). The effect of PRINT CHR$(1) is to make the computer switch to graphics for a code in the range 64 to 95, and this is the effect of pressing the GRAPHICS key. If you look at this second set of graphics closely, though, you'll find that some of the shapes look incomplete. In particular, the face shapes seem to have a slice taken out of the right-hand side. This is because the computer, when it is switched on, defaults to a 'text screen'. In other words, it automatically sets up the screen so as to print words and numbers, rather than graphics symbols. Since the letters and digits do not need so much memory for each character, they can be displayed fully, but the graphics shapes cannot. You can get round this by switching to a different screen layout, one which allows fewer characters per line, but which displays the graphics characters fully. This is done by typing SCREEN 1. Try it before you run the program of Fig.

7.1, and see the difference. From now on, then, each graphics program will use SCREEN 1 (or one of the others which we'll come to later) in place of the 'text screen', which is SCREEN 0. If you want to switch back to the text screen, you need only type SCREEN 0 (then RETURN). A SCREEN command will have the effect of clearing the screen as well as changing it, so CLS isn't needed after a SCREEN 1 or SCREEN 0.

You can do some ornamental work with these shapes if you use the grid of Fig. 7.2 for planning. It shows 32 squares across the screen because it is

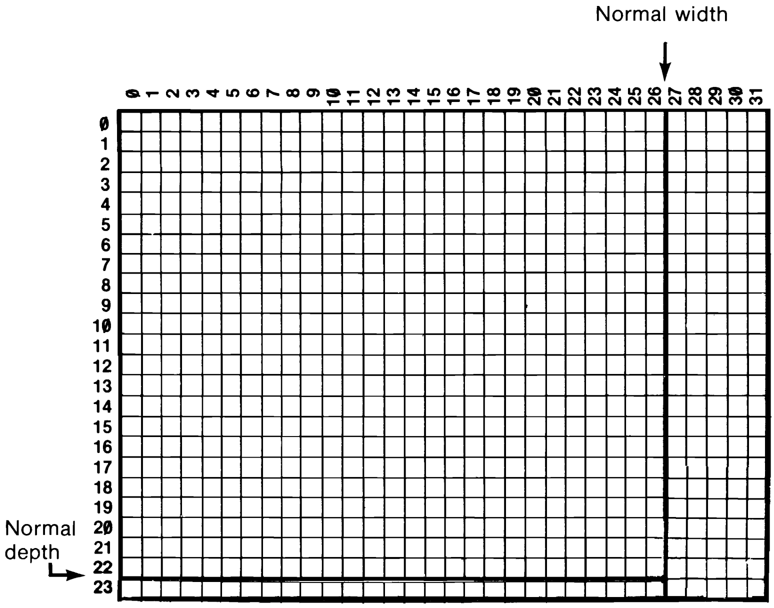

Normal width
Normal depth
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16 17 18 19 2Ø 21 22 23 24 25 26 27 28 29 3Ø 31
Ø 1 2 3 4 5 6 7 8 9 1Ø 11 12 13 14 15 16 17 18 19 2Ø 21 22 23

Fig. 7.2. A planning grid for the graphics shapes.

possible to choose to have up to 32 characters of screen width with SCREEN 1. You get 27 characters per line each time you first select SCREEN 1 on the MSX computer, but you can make this 32 by typing: WIDTH 32. You can, incidentally, also alter the text screen to up to 40 characters per line with the WIDTH command. The choice exists so as to allow you to use practically any TV with the MSX computer, including ones which put the 32nd character on the SCREEN 1 display right at the edge of the screen! You will have to find out for yourself what limits of width you can use. If you are in any doubt, simply leave it alone, and the computer will select 37 characters per line for SCREEN 0 and 27 characters per line for SCREEN 1.

Each square in the grid is the position for a character, and if you draw what you want on a piece of tracing paper placed over this grid, then you can plan what the shape will look like on the screen. There are three ways of programming this. One is to print each line of shapes separately. Another way is to print in a loop, using code numbers that are stored in a DATA line. A third way is to place all of the characters into a string, just as you can type words into a string.

Yes, an illustration would help. Figure 7.3 shows a design, and how it is planned. It might be an emblem which you want to show on the screen. Now

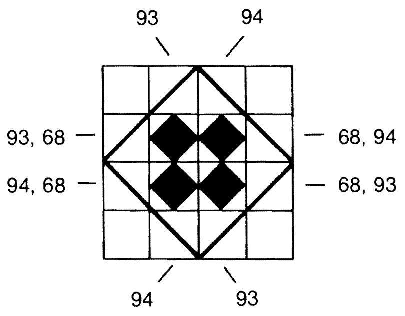


Fig. 7.3. A design which uses the graphics shapes.

you can simply write a program which prints each CHR$ value in the right place, as Fig. 7.4 shows. This works, but it's clumsy programming, because you have to type CHR$(1) so many times. You can make this easier by using

```
10 SCREEN 1
20 CLS:PRINT:PRINT
30 PRINTTAB(5);CHR$(1);CHR$(93);CHR$(
1);CHR$(94)
40 PRINTTAB(4);CHR$(1);CHR$(93);CHR$(
1);CHR$(68);CHR$(1);CHR$(68);CHR$(1);
CHR$(94)
50 PRINTTAB(4);CHR$(1);CHR$(94);CHR$(
1);CHR$(68);CHR$(1);CHR$(68);CHR$(1);
CHR$(93)
60 PRINTTAB(5);CHR$(1);CHR$(94);CHR$(
1);CHR$(93)
```

Fig. 7.4. A simple program to produce the shape.

one of the F-keys to give you CHR$(1), but it still looks clumsy on the screen. Now take a look at Fig. 7.5. This may not look neater to you – it needs more lines, for example, but it is better. There is only one PRINT TAB(4);CHR$(1);CHR$(K%) instruction, instead of three lines of them. Two loops are used, one for each line of characters, and another loop for each column. All of the number variables are integer variables (using the %

```
10 SCREEN1:FOR J%=1 TO 4
20 FOR N%=1 TO 4:READ K%
30 PRINTTAB(4);CHR$(1);CHR$(K%);
40 NEXT:PRINT:NEXT
50 DATA32,93,94,32
60 DATA93,68,68,94
70 DATA94,68,68,93
80 DATA32,94,93,32
```

Fig. 7.5. A neater method, using a loop.

sign) so that the program can run fast. The advantage of this method is that you can see the data clearly, and it's easy to alter the data while keeping the program the same. Note that I've used TAB(4) in each line, and this has meant putting in blanks – CHR$(32) – to pad out the first and last lines. The use of CHR$(1) has no effect when it is followed by CHR$(32).

Figure 7.6 illustrates an even better method, however. It starts by defining a string called B$. This consists of four characters whose code is 29. If you

```
10 SCREEN1:B$=STRING$(4,29):GR$=""
20 FOR J%=1 TO4:FOR N%=1 TO 4:READ K%
30 GR$=GR$+CHR$(1)+CHR$(K%):NEXT
40 GR$=GR$+CHR$(31)+B$:NEXT
50 PRINTTAB(4);GR$
60 DATA32,93,94,32
70 DATA93,68,68,94
80 DATA94,68,68,93
90 DATA32,94,93,32
```

Fig. 7.6. Placing all of the graphics characters and the cursor codes into a single string.

look this up in the Manual, you'll see that it is the cursor left character. The effect of printing B$, then, will just be to put the cursor four places to the left. In line 10 also, the string GR$ is equated to a blank. The next thing is to start two loops, one for the lines, another for the columns. After a line of data has been read and added to GR$ in line 30, CHR$(31) is added. This will cause the cursor to move down one line. Then B$ is added, which causes the cursor to move four spaces left. The total effect, then, is to print four characters, and then move the cursor to the correct position in the next line. Each line is added to the string, and then the complete string is printed in line 40. When you enter this,incidentally, you can save yourself some time if you already have the program of Fig. 7.5 in the memory. Just type:

```
DELETE 10–40
```

and press ENTER. This will remove the old lines 10 to 40, leaving the DATA lines 50 to 80, so that you don't have to type them again. You can then renumber them as 60 to 90 to use in the new program. To do this, type RENUM 60,50,10 and press RETURN.

The great advantage of the method that is illustrated in Fig. 7.6 is that the shape can be printed anywhere on the screen without anything special having to be added to the program. Any PRINT GR$ instruction will print the shape, placed wherever the cursor starts out. You have to be careful, of course, that you don't place the cursor too far over to the right, or too near the bottom of the screen. Armed with this ability to produce patterns, let's see now how we can make them appear in colour.

Vivid impressions

The best place to start on our exploration of colour is with the character shape that we have been using. Figure 7.7 uses the same program to create

```
10 SCREEN1:B$=STRING$(4,29):GR$=""
20 FOR J%=1 TO4:FOR N%=1 TO 4:READ K%
30 GR$=GR$+CHR$(1)+CHR$(K%):NEXT
40 GR$=GR$+CHR$(31)+B$:NEXT
60 DATA32,93,94,32
70 DATA93,68,68,94
80 DATA94,68,68,93
90 DATA32,94,93,32
100 PRINT:PRINT:PRINT
110 COLOR 11,12
120 FOR X%=0 TO 23 STEP 6
130 PRINTTAB(X%);STRING$(5,30);GR$:NE
XT
150 GOSUB 1000
170 FOR BG%=0 TO 15:PRINT"BG= ";BG%;C
HR$(30)
180 COLOR 4,BG%
190 GOSUB 1000:NEXT
200 GOSUB 1000
210 FOR FG%=0 TO 15:PRINT"FG= ";FG%;C
HR$(30)
220 COLOR FG%,1:GOSUB 1000:NEXT
300 END
1000 TIME=0
1010 IF TIME<100 THEN 1010
1020 RETURN
```

Fig. 7.7. Using the COLOR command to change foreground and background colours.

the shape GR$, and then prints a set of four shapes across the screen. This is done in line 130, and the reason for printing STRING$(5,30) is to get the cursor up the screen in the right position for printing the next shape. Lines 170 to 190 then demonstrate how we can change the colour of the whole screen, the background, by itself. Each colour is assigned to a number, and the screen is forced to take the colour corresponding to the number when the COLOR 4,BG% instruction is carried out. The numbers that are assigned to the colours are shown in Fig. 7.8. The program runs through all of the possible background colours, with the pattern displayed always in its dark blue colour, colour 4. It's at this stage that you really need a colour TV to show the results, but you may be disappointed in some of the colours. Red in particular always gives a very 'smeary' appearance on a TV screen, and to see the colours as crisp and clear as they can be, you need to use a colour monitor. As usual, something that is correctly designed for a job is always better than something that is not. TVs are for soap operas; colour monitors are for computer graphics.

Number	Colour
0	Clear
1	Black
2	Green
3	Light green
4	Dark blue
5	Light blue
6	Dark red
7	Sky blue
8	Red
9	Bright red
10	Yellow
11	Light yellow
12	Dark green
13	Purple
14	Grey
15	White

Fig. 7.8. The numbers that are used to produce colours.

The lines 210 to 220 then run through the range of foreground colours, with the background colour set to black. Once again, you will find that some colours appear much more satisfactory than others. Retuning the TV can help a little. Notice that colour 0 is always invisible, and when you make the foreground and the background colours have the same value, all you can see is a blank screen of that value. A very quick way of making a pattern disappear, for example, is to switch its colour to the colour of the background. Notice, by the way, that COLOR affects everything on the screen, whether it was printed before or after the COLOR command. If you find yourself with a screen colour that makes it difficult or impossible to see a listing, then the computer has a 'panic button'. Pressing key F6 (SHIFT F1) restores normal colours, and you don't have to press RETURN to activate the command.

Pixel patterns and high resolution

Up to now, we have produced text on the what is called the *text screen*, SCREEN 0, or on the low resolution graphics screen SCREEN 1, by using the PRINT instruction. We can produce a letter or graphics shape in two ways. Taking A as an example, we could use PRINT "A", or we could use PRINT CHR$(65). The first method is available for the characters that you can see marked on the keys, and for the characters that can be obtained along with the CODE and GRAPHICS keys, but the second method can be used for a larger range which includes the 'control' characters that can shift the cursor, or delete part of a line. The MSX computer, however, allows us to place both letters and graphics characters on to another two varieties of screens, called *graphics screens*.

The differences are important, and the sooner that you can get used to them the better. The text screen (SCREEN 0) is the one that you see when you switch the machine on. It is used, as the name suggests, mainly for text. If all that you want to do is to display figures or words, then the text screen is ideally suited. For the built-in graphics characters, SCREEN 1 is better, because there is more space for each character and bits don't get chopped off. For most of the graphics commands, however, including much more advanced graphics than we have looked at so far, there are two more of these SCREEN numbers that we can use. If we want to make letters and graphics characters appear on these screens, then we need a different method, because PRINT cannot be used. Of the two graphics screens, the one that can be used for the highest resolution of graphics is called screen 2. It can be made to appear simply by including the command SCREEN 2 in your program, but you cannot simply type SCREEN 2, then RETURN, and expect to see it appear. The reason is that the computer *always* switches back to SCREEN 0, or SCREEN 1, whichever was previously in use, whenever a program or command is finished. The only way that you can get enough time to look at the graphics screen is by causing a delay before the program ends. This can be done most easily by programming an *endless loop*, a line like:

```
50 GOTO 50
```

With that in mind, take a look at the program of Fig. 7.9. It starts by

```
10 SCREEN2:COLOR0,4,5:CLS
20 OPEN"GRP:" AS 1
30 FOR N%=0 TO 15
40 COLOR N%,4,5
50 PRINT#1,CHR$(N%+65);
60 NEXT
70 GOTO70
```

Fig. 7.9. Using the high resolution screen, with the COLOR instruction.

calling up the high resolution graphics screen, using SCREEN 2. This is then followed by a COLOR statement, and CLS. The COLOR statement uses *three* numbers this time. Of these, the first is the foreground colour, the second is the background colour, and the third is the border. The border is the outside portion of the screen, where we don't usually place any text or drawings. It can be used in the text screens also, and if you use a small value of WIDTH, you can make the border as big as you like. Getting back to the program, the next line uses OPEN"GRP:" AS 1. This is a way of allowing text letters or numbers to be placed on the high resolution graphics screen. OPEN is a command that we shall look at in detail in Chapter 11. It is normally used for recording data on to cassettes, or reading data from cassettes. In this case, it makes connections that allow text to be sent to the graphics screen. By typing "GRP:", we specify that we want to use the graphics screen. The AS 1 section means that we will use the number 1 as an identifier for this connection. We cannot use PRINT with the graphics screen, but when we specify that we want to print to channel number 1, the computer will look for an OPEN command which uses this number. In this example, it will find that channel number 1 means the graphics screen. The instruction to print on this screen, then, is PRINT #1,"TEXT", with whatever you want to print placed between the quotes.

The next part is a loop, which uses numbers 0 to 15. These are the colour numbers, and they are used in line 40 to set the foreground colour on each pass through the loop. In line 50, we print characters. Because we have used PRINT#1,CHR$(N%+65), these will be the letters of the alphabet, because ASCII code 65 is the code for A. What you will find unexpected is the different colour of each letter! Run this, and just look at it.

Obviously, the high resolution graphics screen does not behave like the text screen! For one thing, each letter can be printed in a different colour, controlled by the foreground colour that is selected in the COLOR command. Another curious point is that if you run this, stop it with CTRL and STOP, and then run again, you will find that the letters appear at a different place along the top line. This is because the high resolution graphics screen (HRG screen, to avoid so much typing!) does not use the ordinary text cursor. You can imagine that there is a graphics cursor, but it is invisible. Unlike the text cursor, it is not placed at the top left of the screen by the CLS or SCREEN commands. The result is that the second set of letters appear to start where the first one left off, but the cursor then returns to the left-hand side for the next set. To make the letters appear starting from the left-hand side each time, put the command PRESET(0,0) between the SCREEN and the COLOR commands in the first line. PRESET affects the 'graphics cursor', and we are just about to move on to that topic.

PSET graphics

The MSX computer offers another way of producing graphics, however. These are now high resolution in the sense that they use very small blocks, or, to give them their proper name, *pixels*. The pixels of the SCREEN 2 are, in fact, the smallest units that we can place on to the graphics screen. We can place up to 256 pixels across the screen, and up to 192 down the screen, a total of 49152 pixels. The MSX computers allow you to specify the colour of each of these pixels, but there are snags. The main snag is that you will *not* get pixels which are immediately next to each other to appear in several different colours. The pixels are grouped in sets of eight across the screen. In any group of eight you are allowed only two colours, one background and one foreground. If you attempt to use a third colour for either background or foreground, the other pixels in the group will turn to this colour.

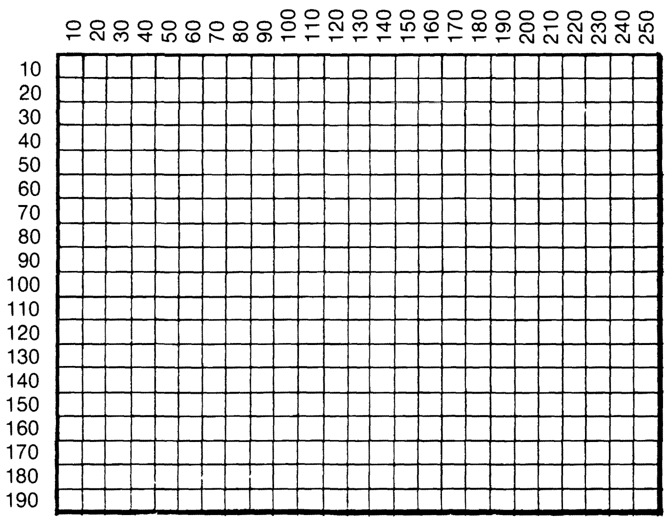
The key instructions now are PSET and PRESET. PSET has to be followed by two numbers within brackets, and its effect is to make a pixel appear in a selected place. By adding another number, outside the brackets, we can also select a colour. If this colour code is omitted, the pixel will appear in whatever foreground colour was selected by the COLOR command earlier in the program. The position of the pixel is specified by two numbers. The first of these, called the X co-ordinate, is the number of units across from the left-hand side of the screen. The screen is divided (in our imagination) into 256 units across and 192 down. We can use numbers 0 to 255 to control the position across the screen, the X-position. We can use numbers of 0 to 191 to control the position down the screen, the Y-position. X=0 means the left-hand side, and Y=0 means the top of the screen. These are very tiny pixels, as you can see from the program in Fig. 7.10. This sets SCREEN 2, then the colours, with a CLS to make the screen change colour. In line 20 a loop starts which will print pixels in a line across the screen. By choosing 95 as the Y-number, we will make this line appear about half-way down the screen. Using N% for the X-number allows us to PSET a number of positions, 15 units apart. The distance apart is measured from the left-hand side of each pixel. Lines 50 to 70 show the effect of using STEP 2. The pixels appear almost joined, and the line of pixels takes noticeably longer to draw.

The effect of PRESET is, as you might guess, to 'reset' the pixel, changing it to background colour so that it disappears. This is used mainly to make pixels appear to move, and we'll look at that point later on.

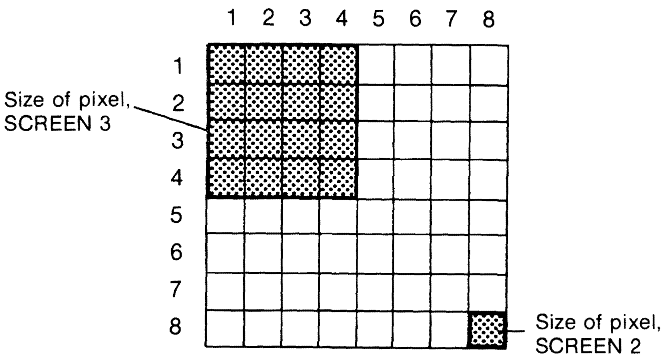
It's time for another example. The main use of PSET and PRESET is in drawing graphs, so that's what we'll illustrate. Figure 7.11 shows, to start with, a PSET-PRESET planning grid, with the numbers 1 to 255 and 1 to 190 to indicate the positions of each pixel. We ought to use 0 to 255 and 0 to 191, but this makes the graph awkward to draw. The important point is that you can draw this grid for yourself. If you buy a pad of graph paper which is scaled in cms and mms, then you can put the numbers on to each sheet for

```
10 SCREEN2:COLOR11,1,13:CLS
20 FOR N%=0 TO 255 STEP 15
30 PSET(N%,95),9
40 NEXT
50 FOR N%=0 TO 255 STEP 2
60 PSET(N%,120),4
70 NEXT
80 GOTO80
```

Fig. 7.10. Lighting up pixels with PSET. This shows how small the pixels are on the high resolution screen.



(a)



(b)

Fig. 7.11. (a) A PSET-PRESET planning grid. (b) Detailed section of the main planning grid.

yourself. You can then shade in the squares that you need to PSET, and so work out the numbers that you need to use. It's even easier if you number the lines of the graph, and represent each pixel position by the places where the lines cross, rather than the squares themselves. Figure 7.12 shows a graph-drawing program. This draws several graphs at the same time, using

```
10 SCREEN2:COLOR 4,11,11:CLS
20 FOR X%=0 TO 255
30 PSET(X%,96+SIN(.1*X%)*90),1
40 PSET(X%,96+SIN(.1*X%)^2*90),9
50 PSET(X%,96+SIN(.1*X%)^3*90),4
60 NEXT
70 GOTO 70
```

Fig. 7.12. A graph-drawing program. Graphs do not look very effective in low resolution.

different colours. Because the pixels of the high resolution screen are so small, however, it's not easy to see the colours of the dots. You can also see that where several dots are very close to each other, they all appear in the same colour. This is the effect of the limitation that only one foreground and one background colour can appear in a group of eight pixels. Line 20 starts the loop which makes use of all the permitted values of X%. The graph shapes are achieved by using the SIN function, with one used as it is, one squared, and one cubed. The multiplying factors are put in to make the shape fill a reasonable amount of the screen in the Y direction. The sine or cube of the sine of an angle cannot have a value less than −1 or more than +1, so we have to 'amplify' it a bit by multiplying by 90. The square cannot have a value of more than +1 or less than 0. The value of X% has to be multiplied by .1 to make the range of angles suitable. The MSX computer does not use angles in units of degrees. Instead, it uses a more natural unit, the radian. One radian is about 57 degrees. The program has an endless loop in line 70 to prevent the text screen from reappearing to spoil the picture, so you will have to press the CTRL and STOP keys together to stop the program.

Sometimes, instead of specifying the exact position on the screen by means of X and Y numbers, you just want to specify a shift, or 'displacement' of a number of pixels. You can do this by using STEP X in place of X and STEP Y in place of Y. Any of the instructions that make use of X and Y (usually in the form of X% and Y%) can use STEP X% and STEP Y% instead.

Lines, boxes, circles and paints!

PSET and PRESET have their uses, when you may want to use a few pixels. It would be hard work, however, to design a program which used PSET and

PRESET to draw lines. Fortunately, the BASIC of the MSX computer allows you to draw lines, boxes, and circles without having to resort to any special effort. This is because of the use of the LINE and CIRCLE commands.

The LINE command, used at full power, can be quite a lot to take in, so we'll start simply. Try the program in Fig. 7.13. This draws a diagonal line, using the small pixels that you should have become used to by now. The

```
10 SCREEN2
20 LINE(10,10)-(240,180),11
30 GOTO 30
```

Fig. 7.13. How the LINE command is used to produce a straight line.

LINE command is followed by two sets of numbers. The first pair, in brackets, are the X and Y numbers for the starting point of the line. By using X=10 and Y=10, we have chosen a position very near the left-hand side and the top of the screen. After the second bracket, there must be a hyphen sign (-). This is followed by the finishing point of the line, in another set of brackets. This uses numbers X=240 and Y=180 to ensure that this point is near the bottom of the screen and at the right-hand side. The result is a diagonal line from top left to bottom right. How about drawing for yourself a line from top right to bottom left?

Now take a deep breath, because there are a lot of extras that can be tacked on to this command. Special offer number one is that once you have drawn one line, you can make the LINE commands simpler. Suppose that you want to draw another line which starts where the first one left off. You don't have to type the starting position all over again; simply omit the first bracket. Figure 7.14 shows what is needed, with line 30 containing LINE-

```
10 SCREEN2
20 COLOR1,1,1:CLS
30 LINE(10,10)-(240,180),11
40 LINE-(10,150),11
50 GOTO50
```

Fig. 7.14. Shortening the LINE instruction for joined lines.

(10,150),11 causing another line to join on to the end of the first one. You must not omit the colour command in this LINE, because if you do, the computer will use the colour which was specified in the COLOR statement - and that's black! This extension to LINE is particularly useful if you want to draw squares - and for random patterns it's essential. Just try Fig. 7.15, which draws a starter line, and then uses a loop in which random numbers are used to place the finishing point of the next lines. You'll see, incidentally, just how fast the MSX computer draws these lines when you run this one.

```
10 SCREEN2:COLOR1,1,1:CLS
20 LINE(20,20)-(150,150),9
30 FOR N%=1 TO 50
40 X%=RND(1)*255:Y%=RND(1)*191
50 LINE-(X%,Y%),9
60 NEXT
70 GOTO70
```

Fig. 7.15. A random lines program.

You could exhibit these at the Hayward Gallery and make your fortune, incidentally, if they weren't so well-drawn.

The next one is quite an astonishment. Try the program in Fig. 7.16, in which the letter B has been added after the rest of the LINE command. The effect is to draw a box – hence the letter B. When you want to draw a box in

```
10 SCREEN2:COLOR11,1,4:CLS
20 LINE(30,30)-(210,140),13,B
30 GOTO30
```

Fig. 7.16. Drawing a box with the LINE instruction.

this way, you must either use the colour number, or the correct number of commas, then the B. You cannot place the B immediately following the last bracket, because this will not be taken as a box command when it is in the place where the computer expects to find a colour command. If you want your box to be in the same colour as the other foreground (colour 11 in this case), then you have to make the command look like: LINE (30,30)–(210, 140),,B. The colour number is omitted, but its comma is not.

The two points in the LINE command form the opposite corners of the box, so you will always get neatly rectangular boxes when you use this command. If any of the sides looks bent, it's time to get your TV serviced! Figure 7.17 shows something of the speed of this command. It chooses two sets of X and Y numbers at random, and then draws a box in a random colour. The number 3 has been added to the random number to make sure

```
10 SCREEN2:COLOR11,1,4:CLS
20 FOR N%=1 TO 10
30 X%=RND(1)*255:Y%=RND(1)*191
40 X1%=RND(1)*255:Y1%=RND(1)*191
50 LINE(X%,Y%)-(X1%,Y1%),3+RND(1)*13,
B
60 NEXT
70 GOTO70
```

Fig. 7.17. A random boxes program.

that none of the boxes is drawn in transparent or black, and RND(1)*13 is used to make sure that the random number for colour does not exceed 15. The colour number ignores fractions, so that if RND(1)*13 gave 12.99 and we added 3 to get 15.99, then the computer takes this as being 15.

No, we haven't finished, because there is one more twist to LINE. Take a look at the simple program in Fig. 7.18. This draws two boxes, using LINE in

```
10 SCREEN2:COLOR4,1,5:CLS
20 LINE(10,10)-(100,100),11,BF
30 LINE(150,20)-(250,190),7,BF
40 GOTO40
```

Fig. 7.18. Filling the box with colour, using the F addition.

the way that you have seen earlier, but with F added to the B. You don't need any commas or other dividers here, just the F at the end. Now the effect of the F is to fill the rectangle with colour. The colour that is used is the colour that you have specified in the LINE command, not the colour that is used for the other foreground drawing. You'll see from Fig. 7.18 that more than one box can be drawn and filled in this way. Figure 7.19 shows how this can be

```
10 SCREEN2:COLOR1,1,1:CLS:A%=RND(-TIM
E):FORN%=1 TO10
20 X%=RND(1)*255:Y%=RND(1)*191
30 X1%=RND(1)*255:Y1%=RND(1)*191
40 LINE(X%,Y%)-(X1%,Y1%),3+RND(1)*13,
BF
50 GOSUB 1000:NEXT
60 GOTO 60
1000 TIME=0
1010 IF TIME<100 THEN 1010
1020 RETURN
```

Fig. 7.19. A random box and fill program to show how one box will cover another.

used in a random box and fill program. Line 10 contains the instruction:

A%=RND(−TIME)

This is the 'seeding' expression for random numbers which we have used before to avoid generating the same sequence each time the program runs. TIME is a number that is read from the internal 'clock' of the MSX computer, and by using this number as a negative number with RND, we ensure that the sequence of numbers that we use in the program does not repeat. If you omit this step, you will see the same boxes being drawn each time you run the program. RND is not quite as random as it should be unless you take this extra step.

Moving in better circles

Drawing straight lines and boxes is useful, but being able to draw circles greatly extends our artistic range. MSX computers, as you might expect by now, have a very useful CIRCLE instruction. As usual, we'll keep it simple for starters. CIRCLE has to be followed by a pair of co-ordinate numbers, in brackets, and then by another number outside the brackets. As usual, commas separate the numbers. The co-ordinate numbers are of the centre of the circle. The number that follows the brackets is the radius of the circle. In case you've forgotten, that's the distance from the centre to the outside. It's measured in screen units, these 256 by 192 units that we work in all the time. If you want to show the whole of a circle on the screen, the largest number that you can use for the radius is 95, assuming that the centre of the circle is the centre of the screen. Following the radius number we can, if we like, have another number, the colour number for the circle.

After that introduction, take a look at Fig. 7.20. Line 10 sets up the familiar SCREEN 2 conditions, and the loop that starts in line 20 causes a set of circles to be drawn. You may find that they don't look very circular.

```
10 SCREEN2
20 FOR N%=10 TO 80 STEP 20
30 CIRCLE(127,96),N%,11
40 NEXT
50 GOTO50
```

Fig. 7.20. The CIRCLE instruction in action.

This is something that depends very much on how well adjusted your TV is. If your TV has a HEIGHT control *outside* the cabinet, try adjusting it until the circles look more like circles. An alternative to this is to adjust the WIDTH control, but few TV receivers nowadays have an adjustable width control, one that you can adjust for yourself. A lot of modern TV receivers have no controls that you can adjust apart from brightness, colour and contrast, and the controls for width and height are inside the cabinet. You must not *on any account* attempt to adjust internal controls unless you know exactly what you are doing. TV receivers are full of high electrical voltages, and only a service engineer knows exactly how to avoid trouble.

We're not finished with circles, though. What you know of MSX computers so far might lead you to believe that there could just be more to this CIRCLE command. There is. Try Fig. 7.21, which shows how you can draw part-circles! The key to this is the provision of start and stop numbers. The number 0 is taken as the 3 o'clock position on the screen, and the circle is

```
10 SCREEN2
20 CIRCLE(127,96),80,11,0,3.14
30 GOTO30
```

Fig. 7.21. Drawing partial circles.

drawn, going *anti-clockwise* from this position. The end-point is specified by the second number. I have made this 3.14, which is the value of PI. Using this number gives a semicircle and, for other parts of a circle, just use the appropriate fraction. Figure 7.22 shows how you can design your part-circles for yourself.

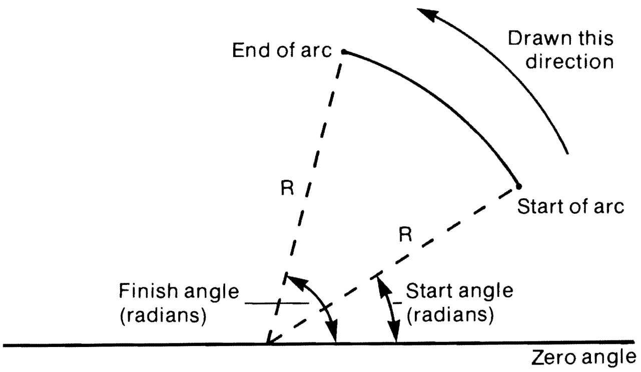


Fig. 7.22. How to design part-circles.

Don't square it, squash it!

MSX computers also allow you to draw shapes which are ellipses – squashed circles. The reshaping of a circle is done by adding yet another number to the circle instruction. If this number is 1, then we simply get a circle. If this extra number is less than 1, however, we get an ellipse which is wider than it is high. If the extra number is greater than one, the ellipse is higher than it is wide. We can even make ellipses which are stretched out so much that they look like straight lines! We can also correct the shape of circles that look elliptical because of the TV receiver. This gives the MSX computer unparalleled power to create all sorts of curved shapes. One of the features of the CIRCLE command is that it allows the use of numbers which take the cursor beyond the screen area, so that what you see on the screen is only part of a drawing.

Take a look at Fig. 7.23. Lines 10 to 60 are used to illustrate the ellipses that we can create. The range of the number that we can use, following the radius number (don't forget the comma) is 0 to 255, but the range that I have illustrated here is the most useful part. The only problem with this command is that it comes after the start and finish commands which we use to draw a part-circle. If you want a complete ellipse, you won't want to use these numbers. We can get round this, as the program indicates, by omitting the start and finish numbers, but *putting in their commas*. It makes the

```
10 SCREEN2
20 FOR E=1 TO .1 STEP -.1
30 CIRCLE(128,96),100,11,,,E:NEXT
40 GOSUB 1000:CLS
50 FOR E=1 TO 3 STEP .2
60 CIRCLE(128,96),80,11,,,E:NEXT
70 GOTO70
1000 TIME=0
1010 IF TIME<100THEN1010ELSE RETURN
```

Fig. 7.23. Producing ellipses. This can also be used to correct the shape of a circle, if your TV does not produce perfect circles.

command look rather odd, but it works! Notice how the circles are drawn starting from the two horizontal ends.

Meantime, there's another command to look at. Figure 7.24 demonstrates another amazing feature of the MSX computers, the PAINT instruction.

```
10 SCREEN2
20 CIRCLE(127,96),88,11
30 CIRCLE(127,96),30,11
40 PAINT(127,50),11
50 GOTO50
```

Fig. 7.24. The amazing PAINT instruction.

This will fill a space with colour, providing that you have enclosed the space with lines. Lines 20 and 30 draw two circles, one within the other. The PAINT instruction in line 40 then fills with colour the space between the circles. PAINT needs the usual pair of co-ordinate numbers following it, in brackets. You have to choose these numbers so that they will act as a starting point for the painting operation. They must, therefore, be somewhere *inside* the area that you want to paint. Odd things may happen if you select a point on the edge of the area that you want to paint. You will certainly not get what you want if you pick a point which is outside the area you want to paint! Following the starting point, we've used one other number. This is the number of the colour that we want to use for painting. We don't have much choice about this colour, it has to be the same as the colour we have used as a boundary. Both of the circles are drawn in yellow, so we have to paint in yellow. If you ignore this, you may find that the colour splashes all over the screen.

In this chapter, we have looked at some of the MSX commands that draw geometrical shapes on the high resolution screen. The examples have all been simple ones which were designed to let you see how the commands worked, and encourage you to try variations. In the following chapter, we're going to look at more complicated examples, and also at the 'free-range' drawing methods that MSX computers allow you to use. We'll also look at the multi-colour screen which is created by SCREEN 3. Even Chapter 8 does not exhaust the capabilities of these amazing machines, however, and Chapter 9 is devoted to animated shapes, or *sprites*. Hang on to your hats!

Chapter Eight
DRAW Graphics

Before we get on to the main themes for this chapter, there's another type of graphics screen to look at, SCREEN 3. This is a lower resolution screen, which uses much larger pixels. The pixel size allows only 64 pixels across the screen by 48 down. If you try some of the line and circle commands of the previous chapter using SCREEN 3 in place of SCREEN 2, you will see how much coarser the lines look. The reason for having this type of display is that it permits the whole range of colours to be used. Figure 8.1 shows this in action, using SCREEN 3 to draw a thick line which has a different colour in each large pixel. The program is straightforward except for the use of MOD

```
10 SCREEN3
20 FOR X%=1 TO 255 STEP 4
30 PSET(X%,96),X%MOD15
40 NEXT
50 GOTO50
```

Fig. 8.1. Using SCREEN 3 for more colour choice, but thicker lines.

in line 30. This command is used in the form XMODY, and it means the remainder when X is divided by Y, using integer division. For example, 5MOD2 would give 1, because 5 divided by 2 is 2 with a remainder of 1. Similarly 14MOD5 is 4 because 5 into 14 is 2 and 4 remaining. In line 30, then, using X%MOD15 will give remainders which are equal to the value of X% until X% is 15, when the value becomes 0. The value of X%MOD15 will then increment up to 14, and then switch back to zero when X% reaches 30. The point of this is that the colour number will not exceed 14 no matter what value X% takes.

The reason for the differences between SCREEN 2 and SCREEN 3 is memory. The computer can deal, at any given time, with 64K of memory. Of this, 32K is taken by the BASIC interpreter, the part which allows the computer to be programmed in BASIC. Of the 32K that remains, only about 28K is actually available to you for writing and running a program, because a chunk of memory is reserved for the machine to store quantities that will be needed when a program runs. These are items like the cursor position, cassette data speed, function key commands and so on. On some machines, yet more memory is taken out of this for the screen display!

Fortunately, the MSX machines use a separate section of memory for the screen display. This screen memory is of a fixed size, however. Using high resolution with two colours takes as much memory as using low resolution with sixteen colours, and you can't have high resolution with sixteen colours without using very much more memory. Since the drawing commands of the MSX machines look so much better on the high resolution screen, we'll keep to that one for most of this book. The low resolution screens will come into their own again in Chapter 9, however, when we look at *sprite graphics*. The sprite graphics capability allows you to use a low resolution graphics screen, with its full colour range, as a background. In front of this background, shapes which are called sprites can be superimposed and moved. These shapes are in high resolution, but because they are comparatively small, they don't eat up too much of the memory. The combination of low resolution background along with sprites provides a very effective way of programming animated games and displays.

POINT the way!

There's another command which fits along with PSET and PRESET, and which gives me a chance to show you PRESET in action. The command is POINT, and it's a way of reporting what's going on. POINT gives you the colour of a pixel. It has to be followed by the usual X and Y location numbers, and you can find what it does by using something like PRINT POINT(X,Y) or A=POINT(X,Y). What is printed or assigned to A by these commands will be a number between 0 and 15. It is the colour number for the pixel, so that you can tell whether the pixel is at background or at foreground colour.

Now that description makes it sound quite simple, but it's not quite so simple as it seems, as the program in Fig. 8.2 will illustrate. Remember in this program, and in the next few, that you will need to press F6 to get back

```
10 SCREEN2:COLOR0,1,1:CLS
20 FOR Y%=0 TO 191:PSET(10,Y%),11:NEX
T
30 FOR Y%=0 TO 191:PSET(254,Y%),11:NE
XT
40 K%=1:X%=11:Y%=1
50 PSET(X%,Y%),5
60 IF POINT(X%+K%,Y%)<>1THEN K%=-K%:Y
%=Y%+1
70 PRESET(X%,Y%)
80 X%=X%+K%
90 IF Y%=190THEN END
100 GOTO50
```

Fig. 8.2. A bouncing ball routine to illustrate the use of POINT.

to normal screen colours afterwards. The program sets a black background, and draws a vertical line down each side. A dot is placed at the top left-hand side, and it moves across the screen. This is done by using PSET(X%,Y%),5. The future position of the dot is tested by using POINT, and if this point is background, the point is PRESET, the value of X% is increased, and the new point is PSET. When the wall is found from POINT, then the variable K% is made negative, so that X%+K% will have the effect of subtracting 1 instead of adding 1. This causes the point to move left rather than right. At the same time, Y%=Y%+1 has the effect of moving the point one step down.

What is not quite so expected is the colour finding action of POINT. If you use:

IF POINT (X%+K%,Y%)=11

then the dot simply zips through the 'wall' and disappears! This is because the colour of the 'wall' is affected by the colour of the dot when the dot gets too near. This is also the reason that holes appear in the wall after the dot has bounced. We can get round this problem very simply, as Fig. 8.3 shows. If you make sure that the dot is the same colour as the wall, then the system works nicely. You can also use the IF POINT(X%,Y%)=11 test if you like, it

```
10 SCREEN2:COLOR0,1,1:CLS
20 FOR Y%=0 TO 191:PSET(10,Y%),11:NEX
T
30 FOR Y%=0 TO 191:PSET(254,Y%),11:NE
XT
40 K%=1:X%=11:Y%=1
50 PSET(X%,Y%),11
60 IF POINT(X%+K%,Y%)=11THEN K%=-K%:Y
%=Y%+1
70 PRESET(X%,Y%)
80 X%=X%+K%
90 IF Y%=190THEN END
100 GOTO50
```

Fig. 8.3. How a change of colour allows you to use POINT more effectively.

works now. I have added the command PRESET(X%,Y%) to line 70 to prevent a dot being left at the wall.

In these examples, of course, there was no need to use POINT, because we *knew exactly* where the walls were. In maze games, however, the walls are drawn at random, and you can't put their X and Y numbers so easily into a POINT command. It's then that POINT really comes into its own. Another reason for using POINT is that if you have two objects moving on the screen, it's easier to detect any kind of collision (object to object or object to background) with POINT. Just one POINT test will detect any type of collision, but if you were testing values of X% and Y%, you might have to use a lot of tests.

The moving point in these two examples is very small, just the size of one pixel on the screen. Try the effect, then, of using SCREEN 3 in both of these programs. You will have to make some adjustments to the numbers, because SCREEN 3 uses only 64 pixels across. As well as SCREEN 3 in line 10, then you will need to use X%=15 and K%=4 in line 40, and Y%=Y%+4 in line 60. When you run this, you'll see the effect of the larger pixels, and also that the speed of the action is much greater. Take your pick!

MSX computer drawing

The ability for drawing lines and circles is just the start of the MSX computer's amazing graphics capabilities. We're going to look at another way of drawing now, one which uses the instruction word DRAW. DRAW has to be followed by a string variable name, like DRAW A$ or DRAW GR$, and it's what you put into this string variable that decides what is drawn. Figure 8.4 shows a list of the letters that can be used. What you have

Letter	Use
A	Angle
B	Blank (no trace left)
C	Colour
D	Down
E	Diagonally up and right
F	Diagonally down and right
G	Diagonally down and left
H	Diagonally up and left
L	Left
M	Move (needs two position numbers)
N	Move, then return to original position
R	Right
S	Scale (numbers 1 to 64)
U	Up
X	Execute substring. A semicolon must follow the name of the substring

Fig. 8.4. The command letters for graphics strings.

to do is to chart your drawing in terms of a starting point, then as up, down, left, right, or diagonal movements. The amount of each movement can be small, just one pixel, so that it's possible to make very detailed patterns in this way when you use SCREEN 2. You can also move to a new starting point without drawing a line. The very considerable advantage of using DRAW is that a complete pattern can be put on to the screen by just one simple instruction like DRAW G$.

Now to the nitty-gritty. Figure 8.5 illustrates just how we go about creating a drawing in this way. Line 10 is familiar stuff, but line 20 is new. In this line, a string is defined. It's a funny-looking sort of string which consists of command letters and numbers. The command letters are the letters of the draw commands, and the numbers are the units of screen size. As you know

```
10 SCREEN2:COLOR11,1,1:CLS
20 GR$="BM20,180;C11U100R20D100R20U10
0R20D100"
30 DRAW GR$
40 GOTO40
```

Fig. 8.5. A drawing program which uses DRAW.

by now, these are 0 to 255 in the X direction, and 0 to 191 in the Y direction. The string starts with BM. B means blank and it's used to ensure that no line is drawn, and M means move. The letter M has to be followed by two numbers, which are the X and Y numbers for the place where you want the drawing to start. I have chosen a point near the bottom left-hand side of the screen. Following the BM step, you need to indicate what colour you want for your drawing. This is done by using the letter C, followed by the colour number. In this case, I'm using colour 11. The next parts are movements – 100 up, 20 right, 100 down, 20 right and so on. The string ends with a quote mark as usual.

Now all that we have to do to draw this in line 30 is the command DRAW GR$. It's delightfully simple, but a very fast and powerful way of creating a drawing. It's particularly easy to make repetitive drawings in this way because we can include a sort of subroutine. This is called a *substring*, and Fig. 8.6 shows how it is used. What it amounts to is that you can define a

```
10 SCREEN2:COLOR11,1,1:CLS
20 SB$="U100R20D100R20"
30 XS$="":FOR N%=1 TO 5:XS$=XS$+SB$:N
EXT
40 GR$="BM20,180;C11XXS$;"
50 DRAW GR$
60 GOTO60
```

Fig. 8.6. Using a substring for a repeated pattern.

string which is part of a pattern, then 'execute' this substring inside the main string. In this case, I have illustrated only the substring being used. The calling command is X (eXecute), and it must be followed by the string name and then a semicolon. If you miss out the semicolon, you will get an error message, *but the error will be reported in line 50*, where the DRAW GR$ instruction is. This can be confusing, because the error is not actually in this line, it's just that it's been found when this line was run. In this example, I have used SB$ to contain a simple up, across, down, across, set of instructions. The loop in line 30 then packs five of these patterns into a longer string, and line 40 calls for this string to be called as a substring of GR$. The result is five sets of the pattern on the screen.

Now try something different, using the program of Fig. 8.7. In line 30, in place of FOR N=1 TO 5, this uses FOR N% = 1 TO 8. This packs eight patterns into the string, and it's effect will be to move the drawing off the screen. Try it *without* line 5, though, and that's not what you find when you

```
5 CLEAR 1000
10 SCREEN2:COLOR11,1,1:CLS
20 SB$="U100R20D100R20"
30 XS$="":FOR N%=1 TO 8:XS$=XS$+SB$:N
EXT
40 GR$="BM20,180;C11XXS$;"
50 DRAW GR$
60 GOTO60
```

Fig. 8.7. The need for CLEAR, and how you can draw off-screen.

try to run it! When we use DRAW, we will be using quite long strings, so we have to clear more memory space for strings than the amount which the MSX computer allows. By having CLEAR 1000 in line 5, we allocate a lot more room. Now you can run the program, and you will find that the boxes disappear off the edge of the screen. The DRAW command does not bother about the edges of the screen, and you will not get an error message if your drawing goes off the screen. This can be very useful, because it's easy to forget just how far you have moved from your original starting place when you have programmed a lot of ups, downs, lefts and rights.

Now for a much more elaborate drawing, in Fig. 8.8, that makes use of all of the commands so far. Lines 10 and 20, as usual, set up the screen conditions, and lines 30 to 50 then define the strings. M$ is the main string,

```
10 CLEAR 1000
20 SCREEN2:COLOR0,1,1:CLS
30 M$="BM40,20;C11D10R10D60L10U5D20U5
R200U60L10U10;XC$;D10L60;XD$;L50U20L6
0"
40 C$="U2LU2LU2LU2L2U2L5U2R30D2L5D2L2
D2LD2LD2LD2L"
50 D$="U5LU5LU2LU3L2U2L4U3L4UL2DL4D3L
4D2L2D3LD2LD5LD5"
60 DRAW M$
70 FOR X%=100 TO 200 STEP 50
80 CIRCLE(X%,100),20,11:NEXT
90 LINE(100,110)-(200,110),11
100 GOTO100
200 SCREEN2:DRAWC$
210 GOTO210
```

Fig. 8.8. A more elaborate drawing which needs to be planned. The LINE and CIRCLE commands can be used along with DRAW.

and it starts with BM40,20. Two substrings are used. In this example, C$ is the chimney, and D$ is the dome. Using XC$ and XD$ in the main string therefore draws these details in the correct places. If you aren't pleased with these places, it's easy to move position. All you have to do is to alter the place where the substring is called. Incidentally, I typed this with the CAPS LOCK pressed, because in the graphics strings, a capital L is less likely to be confused with a 1 than a lower-case l.

I have used semicolons after BM40,20, and before and after each substring. The semicolon must be used after the string ($) sign, but it doesn't have to be used in the other positions. I have put in the extra semicolons just to make it easier to read the items in the string by marking out the positions of the substrings. The main body of the drawing is then carried out at an astonishing speed by DRAW M$ in line 60.

The next point in this example is that you can mix the familiar LINE and CIRCLE commands along with the DRAW! The circle commands are used for drawing the driving wheels, because there is nothing in a DRAW string that can do this. The LINE could have been replaced by a DRAW, but it's easier to use LINE in this case, because it needs only one instruction. When you have started with the colour 0 chosen as the foreground colour (0 means invisible!), then you won't see anything drawn unless you specify a colour. This has to be done in the main DRAW string, in the CIRCLE command, and in the LINE command also.

Now for some more DRAW magic. As well as the up, down, left and right

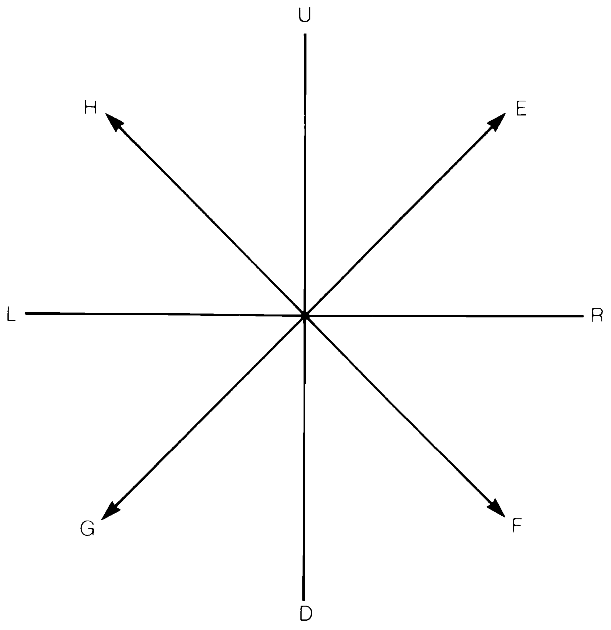


Fig. 8.9. The letters which are used to draw diagonally.

commands, there are letters which indicate diagonal directions. These are illustrated in Fig. 8.9, and a program which uses them is shown in Fig. 8.10. This uses a string which draws a diamond pattern, and then chooses ten randomly selected places on the screen. Now you have to be careful here as to how you get to these places. Using LOCATE works only for the

```
10 SCREEN2:COLOR11,1,1:CLS
20 A%=RND(-TIME)
30 DM$="G10H10E10F10"
40 FOR N%=1 TO 10
50 P%=INT(RND(1)*220+20):Q%=INT(RND(1
)*150+20)
60 PSET(P%,Q%):DRAW DM$
70 NEXT
80 GOTO80
```

Fig. 8.10. Using the diagonal commands in a random diamond program. PSET is used to locate the cursor, because LOCATE does not work with the graphics screens.

instructions of the text screen, like PRINT, but does not move the DRAW position. The command that you have to use is PSET (or PRESET). By picking P% and Q% values at random, followed by PSET (P%,Q%), the invisible graphics cursor is in the correct place to draw the diamond pattern. You might think that you could use "BM P%,Q%" for this, but you can't. You are not allowed to use variable names inside a graphics string, except in ways that we'll come to later. The BM command is not one of the commands that can make use of variables.

Artistic creations

The DRAW command is a very useful way of making straight line drawings with less effort than is needed by LINE. You can use the LINE commands as well, and all the varieties of CIRCLE, along with box fill and PAINT to create any shape you want. What we have to look at now is how to plan these shapes. Trying to make a program that creates shapes on the screen is difficult enough; without planning it's almost impossible! The planning must start, as always, with a piece of graph paper.

You will have to start with a sheet of A4 graph paper. Pads of graph paper made by firms like Chartwell and Guildhall are ideal. They should be A4 size and scaled in centimetres and millimetres. In addition, you will need a pad of tracing paper from the same suppliers. These items are not cheap, but they will last you for a long time. The principle is to mark out on the graph paper the co-ordinate numbers for your graphics screen, place the tracing paper over the graph paper, and then to make drawings on to the tracing paper. Because the tracing paper is transparent, you can see through it to the grid of co-ordinates underneath, and you can read off the values. Figure 7.11

showed the way that you should mark out your graph paper. Strictly speaking, you should use scales of 0 to 255 and 0 to 191, but it's unusual to have to draw right to the edges of the screen, and using 1 to 250 and 10 to 190 is much more convenient – it fits the paper better!

How do we go about designing a DRAW pattern on this? Figure 8.11 indicates how. You count each square on the graph paper as having sides of

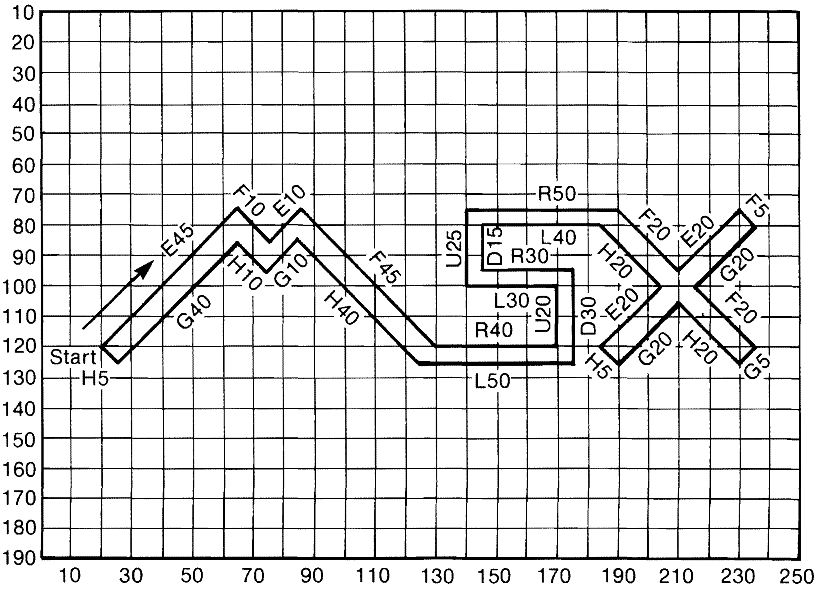


Fig. 8.11. How to use the chart to plan a pattern.

ten pixels, and *diagonals also of ten pixels*. That point about the diagonals is very important, because it saves a lot of awkward calculations or measurements. You draw your patterns on the paper, remembering that you can use up, down, left and right. When it comes to diagonals, remember that these *must* be 45 degree diagonals. This makes some shapes look distorted, like the M in this example. If the distortion is unacceptable, then you will have to use LINE instructions for these parts. When you make the drawing, you will make life a lot easier if you keep to simple dimensions, like multiples of five and ten. It is a lot more difficult to follow a pattern that goes U13L27D17R29 and so on! Working with the tracing paper over the graph paper makes it much easier to see what you are doing, and to check your measurements.

The next step, once the drawing is to your satisfaction, is to obtain any distances and co-ordinates that you need. If you are using LINE and CIRCLE, you will need to read the X and Y co-ordinates of points such as the start and end of a line or the centre of a circle. These are easily read from the graph paper underneath the tracing paper. DRAW graphics are just as easy, and the illustration in Fig. 8.11 shows how. You simply count sides of squares or complete diagonals as ten pixels each, and then write the numbers against the lines. It helps if you write the letters like U,D,G,H and so on as well. You can then program directly from this. Programming is made easier if you program a section at a time, and join the strings up for final drawing. Figure 8.12 shows the MSX shape done in this way. I have used M1$ to hold the upper part of the M, then S1$ for the upper part of the S. The string X$ then holds both parts of the X shape, and strings S2$ and M2$ hold the lower parts of these letter shapes. The whole lot is then put into a string GR$ in line 70, and drawn in line 80. Line 90 then fills the shape with colour, using PAINT. Watch this in action, incidentally, because it gives you a good idea of how complicated the PAINT action is. Note how the paint follows straight line shapes, leaving the corners of the M until last. How about working on your own initials now?

```
10 CLEAR1000:SCREEN2:COLOR11,1,1:CLS
20 M1$="BM20,120;C11;E45F10E10F45"
30 S1$="R40U20L30U25R50"
40 X$="F20E20F5G20F20G5H20G20H5E20H20
"
50 S2$="L40D15R30D30L50"
60 M2$="H40G10H10G40H5"
70 GR$=M1$+S1$+X$+S2$+M2$
80 DRAW GR$
90 PAINT(30,115),11
100 GOTO100
```

Fig. 8.12. The program which has been written from the plan in Fig. 8.12.

Shrink, grow, and turn

The possibilities for creating drawings with the DRAW instruction are made even greater by the options of altering both the size and the angle of patterns. These actions are carried out by using the letters S (for scale) and A (for angle) within the DRAW string. We can also add these instruction letters to a string, and we can put numbers in along with them by using STR$(number). MSX BASIC allows another way of putting variables into these DRAW strings. If you have a variable, such as J%, which carries a value, then you can put something like S=J%; into the string. The *semicolon is essential* and you will get an error message if you omit it. The effects which can be produced with this scale command are spectacular – most other machines could do these actions only with a lot of very complicated programming.

Try the program of Fig. 8.13 now. This, up to line 40, is a simple piece of programming that draws a box. After the delay in line 50, though, things start to happen. The fancy business starts in line 60, with the loop that uses variable J%. The range of the counter J% is 1 to 60, and you can use up to 255. This figure of 255 is the whole of the permitted range for the scale

```
10 CLEAR500:SCREEN2
20 COLOR11,1,1:CLS
30 G$="U50R50D50L50"
40 DRAW"BM128,96"+G$
50 FOR N=1 TO1000:NEXT
60 FOR J%=1 TO 60:CLS
70 DRAW"BM128,96S"+STR$(J%)+G$
80 FOR Z=1 TO 200:NEXT:NEXT
90 GOTO90
```

Fig. 8.13. Using S for scaling a drawing.

instruction, which uses the letter S. The scale which will be used for drawing is one eighth of the number which follows S. For example, if you use S2, then the drawing is 2/8, which is 1/4 size. Using S16 would make the drawing 16/8, which is double size. This letter S has been put into the DRAW instruction in line 70, so that the pattern is drawn with a different scale number each time. Watch these scale effects carefully. They are very easy to use, but the effects may not be exactly what you want. One point is that the start which is given by the BM part of the command is one edge of the square. The square will always start at this edge and grow from that point. The other problem occurs if the pattern 'grows' too much. The reason is that the pattern will not grow beyond the edges of the screen. If your pattern hits a screen edge, it will, from then on, *grow in the other directions* only. This can cause the pattern to shift from where you thought it would be and also to change shape. Some careful planning, with graph paper and tracing paper, is needed when you start to work with these Incredible Hulk graphics! You will also find that if you stop the program, using CTRL and STOP, then when it starts again, it will draw one shape of the size it was drawing when it was stopped. It will then restart normally. This is because the S size is stored in the memory, and unless you clear it with NEW (which will clear out the program also) it will stay put. When you use several drawings in a program, you will need to prevent a scale factor from one from affecting the others. This can be done by including S8 in each drawing that you want to be normal size.

Take a quick look now at a small change which makes a big difference. The program is in Fig. 8.14, and lines 10 and 20 should be familiar territory for you now. Line 30 is a short and simple string for a shape. Notice that this

```
10 CLEAR500:SCREEN2
20 COLOR 11,1,1:CLS
30 G$="BU5L10F5G5R20H5E5L10"
40 DRAW"BM140,80"+G$:FORN=1 TO 1000:N
EXT
50 FOR J%=1 TO 60:CLS
60 DRAW"BM140,80S"+STR$(J%)+G$
70 FOR Z=1 TO 200:NEXT:NEXT
80 GOTO 80
```

Fig. 8.14. Starting a drawing at the centre, so that it expands round the centre.

shape has *no* defined starting point, and it starts with a blank move upwards. The reason, as you'll see later is so that the starting point is the *centre* of the shape. When we expand the shape, the expansion is always around the starting point. If this now is the centre, the centre of the shape stays put. In the previous example, because the starting point was at a corner, the shape expanded from that corner outwards. Line 40 then draws this shape, but with a starting point added at 140,80. Notice how this is done, using the + sign to join the strings.

Angle antics

The use of the command letter S to make the drawing take different scales is a splendid feature of MSX BASIC, but there is another command letter that we can use. This time it's A, and its effect is to alter the angle at which a shape is seen. With A0, the shape is shown just as it has been drawn. With A1, the shape is turned through 90 degrees anticlockwise. Using A2 makes the shape turn through 180 degrees, and A3 makes it turn through 270 degrees. The number which is used with A must not exceed 3, otherwise the program will stop with the 'Illegal function call' error message. Figure 8.15 shows an example of this command in action. The shape is drawn, using S8A0 to

```
10 CLEAR500:SCREEN2
20 COLOR 11,1,1:CLS
30 G$="BU5L10D10R20H5E5L10"
40 DRAW"BM140,80S8A0"+G$:FORN=1 TO 10
00:NEXT:CLS
50 FOR J%=0 TO 3
60 DRAW"BM140,80A"+STR$(J%)+G$
70 FOR Z=1 TO 1000:NEXT:CLS:NEXT
80 GOTO80
```

Fig. 8.15. Using the angle-turning command letter A.

make sure that its size and angle will be unaffected by any previous program that was running. It is then rotated by using values of J% ranging from 0 to 3, with the value put into the graphics string in the same way as before, using STR$. Some MSX machines used with some TV receivers will make the shape appear to alter as it turns. This is the same problem as is manifested by the shape of circles. If your TV can be adjusted so as to show perfectly round circles, it will also show no change in a shape which is being rotated.

Multiple shapes

Now take a look at Fig. 8.16. This illustrates how a number of shapes can be joined up easily. The key to this is the use of the + or − signs along with M or

```
10 CLEAR 500:SCREEN2
20 COLOR 11,1,1:CLS:DRAW"BM10,80"
30 G$="BU5L10F5G5R20H5E5L10"
40 FOR X%=1 TO 8
50 DRAW"BM"+"+"+STR$(20)+",5"+G$
60 FORN=1 TO1000:NEXT:NEXT
70 GOTO70
```

Fig. 8.16. Shifting a pattern position so as to join patterns with the + sign.

BM. Adding the sign + to a BM or M instruction will cause a movement of as many spaces as you specify. This is an important difference. BM 10,10, for example, means move to position X+10, Y+10. If we use BM+10,+10, we mean a move of ten places right and ten places down from where the last piece of drawing finished. Note that if you have been using angle and scale commands, you can sometimes find that + gives movement left or up, and − gives movement right or down. It's advisable to reset all scale and angle commands before you use the + and − markers. The tricky bit here is adding the + or − signs to the string, and line 50 shows two ways of doing this. The + sign which is enclosed by quotes is the one that is put into the string; the others are there only to provide the joining action. The method that uses STR$(20) is more useful when the quantity that is being added is a variable value. The method that uses ",5" is more suitable when the value is a fixed and known number.

Chapter Nine
Identifiable Flying Objects

Animation of a shape on the screen can be a tedious process. We have seen something of it in the 'bouncing ball' program of Fig. 8.2 to know what's involved. You have to print your object at a place on the screen, wait a short time, then wipe out the object. This can be done by using PRESET, or by changing its colour to transparent. You then have to shift the (invisible) graphics cursor, and repeat the process. This has to be done in a loop, with X and Y position values chosen so that the object follows the path that you want. It's bad enough to have to animate a point, but animating a shape with the ordinary commands of your average computer is like hacking salt from the Siberian mines with less pleasure in the work. Nevertheless, this is about all that the average computer can do with BASIC language commands. Fortunately, we're not dealing with a computer that is 'average' in any respect. The MSX computer has the ability to create and control moving objects, called sprites. You can determine the shape, and to some extent the size, of these sprites for yourself. They are controlled by BASIC instruction words, which makes the MSX computer one of a select bunch of computers – for many computers use number codes to control sprites, and this makes it very hard to remember what you have to do. MSX computer sprites can be created and controlled, once you have some practice in the art, without the need to keep the manual in one hand all the time! It's because sprite graphics are available that I have not described animation in any detail until now.

Sprite creation

Working with sprites means that you have to determine the shape and size of the sprites, and then arrange for instructions that will move them. The point of sprite graphics is that you don't have to go through the process of printing and wiping; this is done automatically. Keeping to the policy of one thing at a time, we'll start this chapter by looking at how a sprite is created. As we go on, you'll become more familiar with the ideas of sprites, and you will be able to take on the job for yourself, using your own ideas.

To create a sprite shape, we have to start on paper, and our starting place is the 8 × 8 grid that is shown in Fig. 9.1. You can see that each column of this grid is numbered, starting with 1 on the right-hand side, and ending with 128 on the left-hand side. These numbers are very important, because they decide what your sprite shape will look like. You create a shape by pencilling in squares on the grid. You must shade complete squares, not part-squares. Once you have done this, you can work out a set of 8 code numbers, one for each row of the grid. This is done by looking along a line, and adding up the column numbers for each square that is shaded. If only the square on the right-hand side is shaded, then the number is 1. If only the square on the left-hand side is shaded, then the number is 128. If both of these squares are shaded, the number is 129 – we just add 128 and 1.

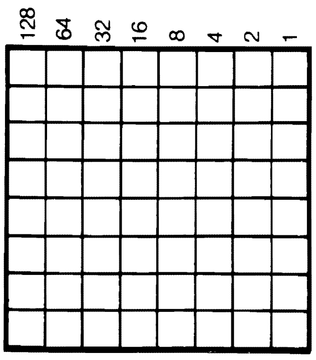
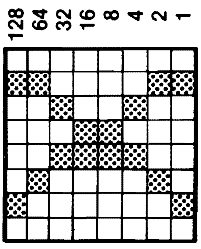


Fig. 9.1. The 8 × 8 sprite planning grid.

What we need now is an illustration, and Fig. 9.2 shows the first step, the 8 × 8 grid drawing, a shape drawn over it, and the set of numbers. The best way of doing this is to place a piece of tracing paper over the grid, and then shade on to the tracing paper. That way, you only have to draw your grid once. It's also a lot easier to change your mind if you use tracing paper. Once the shape is drawn, we can work out the eight code numbers and these have been shown at the side of the drawing, which is of the dreaded Flying Wotsit of Argalia. The next step is to make these numbers into the form of a string. This means using a loop which will read each number from a DATA line, and add the CHR$ of each number to a string variable. I've used the name SP$ for this variable.



Ø
128 + 64 + 2 + 1 = 195
32 + 4 = 36
16 + 8 = 24
32 + 16 + 8 + 4 = 6Ø
64 + 2 = 66
128 + 1 = 129
Ø

Fig. 9.2. How to plan a sprite, and find its eight code numbers.

This, however, doesn't make the shape appear on the screen. This is dealt with by another instruction word, SPRITE$. SPRITE$ has to be followed by a number within brackets. This number is a reference number for this sprite shape, so that you can call it up when you want it. This is the point at which the sprite is actually created. It's still invisible, however, until we deliberately place it on the screen by using a PUT SPRITE command. This command has to be followed by five numbers. The first of these is a *sprite plane*. This is a sort of priority order of sprites, and it's important only if you have more than one sprite. If you have two sprites, one on plane 0 and the other on plane 1, then when the sprites meet, the one on plane 0 will always appear to pass *in front* of the sprite on plane 1. All sprites will appear to pass in front of anything else that is drawn on the screen (the 'playfield'). Following the sprite plane number is a comma, then two numbers in brackets, also separated by a comma. These are the familiar X and Y position numbers. They refer to the screen position in the usual way. A comma follows, then a colour number, because you can have your sprite in any colour that you like. The final number in the PUT SPRITE command is the reference number for your sprite, the number that you gave it in the SPRITE$ command. You can use for background any screen in the range 1 to 3. Only the text screen, SCREEN 0, cannot be used with sprites.

Now take a look at the program in Fig. 9.3. This starts with SCREEN 2,0.

```
10 SCREEN2,0:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 PUT SPRITE 0,(128,96),11,1
60 GOTO60
100 DATA 0,195,36,24,60,66,129,0
```

Fig. 9.3. Creating a sprite and placing it on the screen.

The SCREEN 2 part is the familiar command to select the high resolution screen, but the extra 0 is a new feature. What this does is to select the size of all the sprites on the screen. Using a 0 here selects small sprites, a 1 selects larger sprites. You can also use the numbers 2 or 3 for gigantic sprites, but that needs rather more preparation – we'll look at it later. Line 10 also clears the screen, and prepares an empty string SP$ ready for use. Lines 20 and 30 then fill this string, reading numbers from the DATA line (line 100), converting each into CHR$ form, and adding to the string. In line 40, SPRITE$(1) is then equated to the shape-string, SP$, and your sprite is formed. Line 50 then places the sprite on the screen. We're using plane 0, a position around the centre of the screen, colour 11, and the reference number of 1 which matches the number we used in the SPRITE$ instruction. Line 60 then keeps things steady so we can look at the shape. That's it!

Frankenstein's fun

Now that we have created this object, what about using it. The first thing to try is to move it. We'll start the object off at one corner of the screen and move it from there. Figure 9.4 shows the program modified so as to do this.

```
10 SCREEN2,0:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=1
60 PUT SPRITE 0,(X%,Y%),11,1
70 X%=X%+1:Y%=Y%+1:IF Y%<191 THEN 60
80 GOTO80
100 DATA 0,195,36,24,60,66,129,0
```

Fig. 9.4. Animating a sprite. This is done by changing X and Y position numbers. You don't need any commands for printing or deleting the shape.

To start with, we have to change the PUT SPRITE instruction. By using the editing commands, I changed the line number of this command from 50 to 60. Line 50 now assigns values to X% and Y%, which will be used as position numbers. These are now put into the PUT SPRITE command in line 60. Line 70 then tests for the Y% value reaching 191, and loops back to the sprite command if this value has not been reached. In addition, the values of both X% and Y% are incremented so that the sprite will move diagonally, down and right until it vanishes from sight.

Now is the time to play with this simple program. To start with, try the effect of different numbers in line 70. Instead of Y%<191, try Y%<200. This, as you can see, does not cause any problems. Now for a real surprise, try Y%<1024. Instead of the frantic error messages that you get from some computers, the MSX computer interprets this as an instruction to make the sprite appear four times. The rule here is that a number of 256 corresponds to a complete 'lap' of the screen in whatever direction the sprite is travelling. The number 1024 is four laps of the screen, and that's what it does! It's very neat, and so simple to program.

Interruptions welcome!

Now for something really interesting. So far, we've tied up the action of the computer when we have been using sprites. When you move your sprites by means of a FOR...NEXT loop, you can't have the computer doing anything else. The MSX computer, however, provides for doing two things at once – or so it appears. It's all done by what are called *interrupts*. You can instruct the machine to break off whatever else it's doing fifty times a second, and run a subroutine. (In the USA, the value is sixty times per second.) This subroutine could be a sprite-moving subroutine, and the result will be that the sprites are moved automatically in each interrupt, but that other actions can be carried out in the rest of the time! Take a look at Fig. 9.5. I have shifted the sprite-moving commands to a subroutine which starts at line 1000, and which ends with the usual RETURN. In line 60, now, is the

```
10 SCREEN2,0:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=1
60 INTERVAL ON
70 ON INTERVAL=1 GOSUB 1000
80 GOTO80
100 DATA 0,195,36,24,60,66,129,0
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1:Y%=Y%+1
1020 RETURN
```

Fig. 9.5. Using INTERVAL in sprite animation. This allows the computer to animate the sprite and still do other things!

command INTERVAL ON. This is needed to start the interrupting process. In line 70, we instruct the computer what is to be done at each interrupt. ON INTERVAL =1 means that something is to be done at each interrupt. The 'something' is GOSUB 1000, the line that moves the sprite.

Try it – and be surprised. This time, the movement does not stop after just one lap. The sprite will be moved across the screen each time an interrupt takes place, and the value of X% and Y% will keep increasing. If you want to put a limit on it, Fig. 9.6 shows how. The value of one of the position numbers, Y% in this case, is tested. If this exceeds a fixed value (1024 in this example), then the INTERVAL OFF command stops the interrupts from having any effect. The added line is numbered 1015 in Fig. 9.6. Try also the effect of ON INTERVAL =2 or ON INTERVAL =4 and so on.

Now just in case you're not convinced that INTERVAL allows two things

```
10 SCREEN2,0:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=1
60 INTERVAL ON
70 ON INTERVAL=1 GOSUB 1000
80 GOTO80
100 DATA 0,195,36,24,60,66,129,0
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1:Y%=Y%+1
1015 IF Y%>1024 THEN INTERVAL OFF
1020 RETURN
```

Fig. 9.6. Stopping sprite animation with INTERVAL OFF.

at once, try Fig. 9.7. There are a number of small changes in this program, and they all have an effect. The first point is that we use SCREEN 2,1 to get the larger size of sprite. Then, in line 40, we make a variable N% equal to 0. Line 80 prints the value of this variable, then increments it. Remember that you have to use OPEN"GRP:" AS 1 if you are to be able to use PRINT#1 on the graphics screen. Line 90 puts in a short time delay, and then goes back

```
10 SCREEN2,1:CLS:SP$=""
15 OPEN"GRP:" AS 1
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$:N%=0
50 X%=1:Y%=1
60 INTERVAL ON
70 ON INTERVAL=1 GOSUB 1000
80 PRINT#1,N%:N%=N%+1
90 FOR J=1TO 20:NEXT:GOTO80
100 DATA 0,195,36,24,60,66,129,0
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1:Y%=Y%+1
1015 IF Y%>255THEN INTERVAL OFF
1020 RETURN
```

Fig. 9.7. Yes, you can do two things at once!

to line 80 to keep the printing going. When you run it, you'll get quite a number of surprises. The first one is that the numbers are printed at the position of the sprite. This is because the PRINT#1 instruction will cause a number to be printed at the graphics cursor position, and that's also where the sprite is. Once the sprite has been across once, though, you will see the numbers start to increment on the left-hand side of the screen. All of this proves that the computer is capable of printing the numbers and moving the sprite. When the screen fills with numbers, you get your second surprise. Instead of scrolling upwards as it usually does, the screen stays filled with numbers, and the printing starts all over again on the top of the screen. Furthermore, the numbers print over the odd ones instead of replacing them. The rules about how the screen displays things are quite different when you are displaying sprites in this way! It is designed so that any background that you have drawn for your sprites will not scroll off the screen.

It's the pecking order that counts

One of the features of sprites is that they have a system of priorities. Priority means that you will always see a sprite pass across the screen even when other graphics are present. It appears, in other words, to move in front of the other graphics. On a SCREEN 1 type of display, it can even move in front of text. Figure 9.8 demonstrates this, and also shows that the action of a sprite

```
10 SCREEN1,1:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$:N%=0
50 X%=1:Y%=1
60 INTERVAL ON
70 ON INTERVAL=1 GOSUB 1000
80 FORN=1TO20:PRINT"THE NAME IS SPRIT
E - MSX VARIETY":NEXT
90 GOTO90
99 GOTO80
100 DATA 0,195,36,24,60,66,129,0
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1:Y%=Y%+1
1015 IF Y%>1024THEN INTERVAL OFF
1020 RETURN
```

Fig. 9.8. Illustrating sprite priority over the background.

is not affected by the scrolling of a text screen. The printing is being carried out while the sprite is moving, and the sprite appears to continue to move in front of the letters after printing has stopped. It can be a very effective way to point out the items of your menu!

More important, though, one sprite will appear to move in front of another. Take a look at the program in Fig. 9.9. This creates a blue

```
10 SCREEN2,1:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=96:P%=255:Q%=96
60 GR$="BM110,190;C13S8A0U190R25D190L
25"
70 DRAW GR$
80 PAINT(120,100),13
90 INTERVAL ON
100 SP$="":FOR N%=1 TO 8:READ K%
110 SP$=SP$+CHR$(K%):NEXT
120 SPRITE$(2)=SP$
130 ON INTERVAL=1 GOSUB 1000
140 GOTO140
150 DATA 0,195,36,24,60,66,129,0
160 DATA 24,24,24,255,255,24,24,24
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1
1020 PUT SPRITE 1,(P%,Q%),1,2
1030 P%=P%-1:RETURN
```

Fig. 9.9. Priority of one sprite over another.

background, and then draws a magenta column which is almost the full height of the screen. Two sprites then pass across. One is the Wotsit we have used before, colour yellow, but the other is a black cross. Now both of these sprites, as you would expect, pass in front of the column, but you'll see that the yellow Wotsit also passes in front of the black cross. This is because the yellow Wotsit is a sprite on plane 0, but the black cross is a sprite on plane 1. Plane zero has a higher priority than plane 1, so its sprite always appears in front of anything else. A sprite on plane 1 will be behind a sprite on plane 0, but it will always be in front of a sprite on plane 2, or any higher numbered plane. You can use plane numbers from 0 (top priority) to plane 31 (end of the line), but no more. This makes it possible to have a total of 32 moving (or fixed) sprite objects on the screen at the same time!

Figure 9.10 summarises the rules about sprites so far. It's important to remember that you can't use a sprite until it has been created, so all lines like

1. Sprite shapes should be defined early in a program, using SPRITE$.
2. The PUT SPRITE should also be used before any ON INTERVAL or similar lines.
3. Sprite numbers can be 0 to 255 for small and medium size sprites, 0 to 63 for large sprites.
4. Sprite planes are numbered 0 to 31, and the lower the number, the higher the priority.

Fig. 9.10. A summary of the sprite rules.

ON INTERVAL = 1 GOSUB 1000 must *follow* the lines that allocate SPRITE$ number and shape. There's more to come, though. It's useful to have each sprite on a different plane, but it leaves one problem. Suppose we *want* sprites to collide? We might, for example, have one sprite which was a missile and another which was an aircraft. Now you can have more than one sprite on a plane – but they can't be controlled independently. If, in the program of Fig. 9.9 you alter line 1020 so that the sprite plane is 0, you will see that only one sprite appears. This will be one that was most recently defined, the one in line 1020. If you want to have sprites moving in different paths, you must allocate them to different planes – so how do we get collisions?

The answer lies in another interrupt type of instruction. It has to be prepared for by the instruction SPRITE ON, which we can add to line 90. This allows collisions to be detected by a line that reads:

```
ON SPRITE GOSUB 3000
```

You can, of course use any line number you like for the subroutine. In this subroutine, you can then put in whatever you want to happen when the sprites collide. Suppose, for example, that we just freeze everything by using INTERVAL OFF. Figure 9.11 shows the effect of this. When you run this one, the sprites meet – and then freeze!

```
10 SCREEN2,1:CLS:SP$=""
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=96:P%=255:Q%=96
60 GR$="BM110,190;C13S8A0U190R25D190L
25"
70 DRAW GR$
80 PAINT(120,100),13
90 INTERVAL ON:SPRITE ON
100 SP$="":FOR N%=1 TO 8:READ K%
110 SP$=SP$+CHR$(K%):NEXT
120 SPRITE$(2)=SP$
130 ON INTERVAL=1 GOSUB 1000
140 ON SPRITE GOSUB 3000
150 GOTO150
160 DATA 0,195,36,24,60,66,129,0
170 DATA 24,24,24,255,255,24,24,24
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1
1020 PUT SPRITE 1,(P%,Q%),1,2
1030 P%=P%-1:RETURN
3000 INTERVAL OFF
3010 RETURN
```

Fig. 9.11. How sprites can be made to collide.

If you're not keen on frozen sprites, how about Fig. 9.12. This does rather more when the collision is detected. The sprites which have collided are removed. You must do this before you attempt to put anything else on this piece of the screen, because sprites *always* appear in front of anything else. You don't, in fact, have to move the sprites if you make their colour transparent, but since it's one and the same command, PUT SPRITE, we might as well do both. We can then draw the circle, and print the word SPLAT! in the collision space. This has to be done carefully. If any of the letters of the word come too close to the edge of the circle, they will appear 'smeared'. I have avoided this by using a fairly large circle. If you try a radius of 20, you'll see what I mean.

Bigger and better

Sprite sizes are not confined to the 8 × 8 grid that we have worked with so

```
10 SCREEN2,1:CLS:SP$=""
15 OPEN"GRP:" AS 1
20 FOR N%=1 TO 8:READ K%
30 SP$=SP$+CHR$(K%):NEXT
40 SPRITE$(1)=SP$
50 X%=1:Y%=96:P%=255:Q%=96
60 GR$="BM110,190;C13S8A0U190R25D190L
25"
70 DRAW GR$
80 PAINT(120,100),13
90 INTERVAL ON:SPRITE ON
100 SP$="":FOR N%=1 TO 8:READ K%
110 SP$=SP$+CHR$(K%):NEXT
120 SPRITE$(2)=SP$
130 ON INTERVAL=1 GOSUB 1000
140 ON SPRITE GOSUB 3000
150 GOTO150
160 DATA 0,195,36,24,60,66,129,0
170 DATA 24,24,24,255,255,24,24,24
1000 PUT SPRITE 0,(X%,Y%),11,1
1010 X%=X%+1
1020 PUT SPRITE 1,(P%,Q%),1,2
1030 P%=P%-1:RETURN
3000 INTERVAL OFF
3010 PUT SPRITE 0,(254,254),0,1:PUT S
PRITE 1,(254,254),0,2
3020 XX%=(P%+X%)/2:CIRCLE(XX%,Y%),30,
15
3030 PAINT(XX%,Y%),15
3040 PRESET(XX%-20,Y%):COLOR1,0:PRINT
#1,"SPLAT!"
3050 RETURN
```

Fig. 9.12. Programming a more dramatic collision.

far. The MSX computers provide for mammoth sprites (spritephants?) which are planned on a 16 × 16 grid. This consists of four 8 × 8 grids arranged as shown in Fig. 9.13, and it provides a new challenge. How do we design sprites on this grid?

To start with, we have to draw the shape on the grid. Figure 9.14 illustrates how this is done. As far as the drawing goes, this is much the same as before, but with more squares to shade. What you need to remember carefully is the order in which the four 8 × 8 pieces of the 16 × 16 grid are filled. The program of Fig. 9.15 drives this home. The shapes that are dictated by the DATA lines 100 to 130 consist, in order, of a George cross, St. Andrew's cross, large square, and diamond. When the sprite appears on the screen, you will see these in the positions that are indicated in Fig. 9.14. When you design a large sprite, then, you have to put the lines of data into

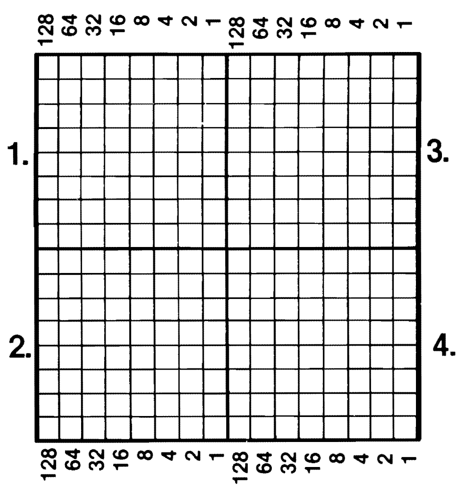


Fig. 9.13. A planning grid for large (16 × 16) sprites.

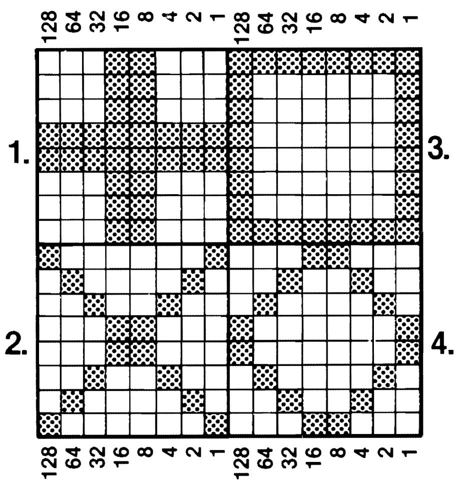


Fig. 9.14. A shape planned on the large grid. You have to be careful of the order in which DATA lines are written.

the correct order. That means the order of top left, bottom left, top right, and then bottom right. Figure 9.16 shows what is involved as far as planning is concerned, and Fig. 9.17 shows a program which will produce this shape. Note how the DATA lines 100 to 130 have been arranged in the order of the four sections of the sprite.

At this stage we have to look at a few more rules. When you use the small sprites, size 0 or 1, then you can use a large range of sprite numbers, 0 to 255 in the SPRITE$(n) instruction. When you use the large sprites, you are restricted to numbers 0 to 63. Since you can't have more than 32 sprite

```
10 SCREEN2,2:SP$=""
20 FOR N%=1 TO 32
30 READ K%
40 SP$=SP$+CHR$(K%):NEXT
50 SPRITE$(1)=SP$
60 PUT SPRITE 0,(128,96),11,1
70 GOTO 70
100 DATA 24,24,24,255,255,24,24,24
110 DATA 129,66,36,24,24,36,66,129
120 DATA 255,129,129,129,129,19,129,2
55
130 DATA 24,36,66,129,129,66,36,24
```

Fig. 9.15. A giant sprite made out of four shapes to show the order of placing the shapes.

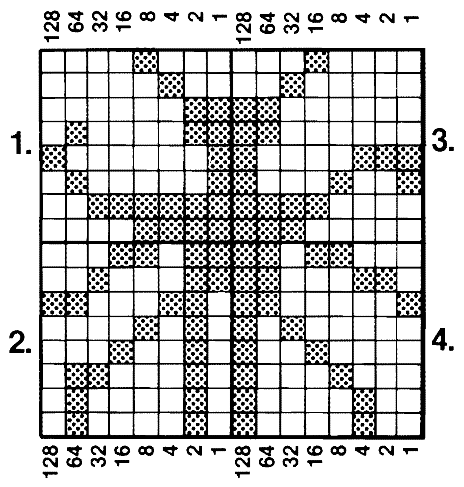


Fig. 9.16. Planning a large sprite shape.

```
10 SCREEN2,2:SP$=""
20 FOR N%=1 TO 32
30 READ K%
40 SP$=SP$+CHR$(K%):NEXT
50 SPRITE$(1)=SP$
60 PUT SPRITE 0,(128,96),11,1
70 GOTO 70
100 DATA 8,4,3,67,129,65,63,15
110 DATA 27,35,198,10,18,34,66,66
120 DATA 16,32,192,192,135,137,240,22
4
130 DATA 216,198,193,160,144,136,132,
132
```

Fig. 9.17. The large sprite program. Try this with SCREEN 2,3 also.

planes, it makes sense to keep to numbers 0 to 31, and you can then use either size of sprite with no worries.

More control

As a final touch, what about more direct control of our sprites? We could use joystick control, and a method for doing this is illustrated in the MSX computer manual. There is another interrupt control ON STRIG GOSUB, which causes a subroutine to be carried out when the spacebar, or a joystick firing button, is pressed. This has to be 'loaded' by having a STRIG command earlier in the program. STRIG has to be followed by a number within brackets, and this number specifies which control carries out the action. STRIG(0) is the spacebar on the keyboard, and numbers 1 to 4 are values for the four trigger buttons on the MSX joysticks. Figure 9.18

```
10 SCREEN2,2:SP$=""
20 FOR N%=1 TO 32
30 READ K%
40 SP$=SP$+CHR$(K%):NEXT
50 SPRITE$(1)=SP$:STRIG(0) ON
60 PUT SPRITE 0,(128,96),11,1
70 ON STRIG GOSUB 200
80 GOTO 80
200 PUT SPRITE 0,(256,200),11,1
210 CIRCLE (128,96),10,5
220 RETURN
1000 DATA 8,4,3,67,129,65,63,15
1010 DATA 27,35,198,10,18,34,66,66
1020 DATA 16,32,192,192,135,137,240,2
24
1030 DATA 216,198,193,160,144,136,132
,132
```

Fig. 9.18. The use of STRIG for spacebar of joystick trigger control.

illustrates this command in action, with a sprite being zapped by pressing the spacebar. The command is 'armed' in line 50 by using STRIG(0) ON. It is then used in line 70 to make the program jump to the subroutine at line 200 when the spacebar is pressed. This subroutine moves the sprite out of sight, and puts a faint circle in its place. Alas, poor sprite ...

Another instruction of the same type is the ON KEY command. This operates rather like ON K GOSUB. You enable it with KEY ON, and then follow this with ON KEY GOSUB 1000,2000,3000... using as many subroutine numbers as you want to use the F keys. Since you have ten F keys (F1 to F10), you could have up to ten subroutines here. When you press F1, the first subroutine in the list will run; when you press F2, the second subroutine will run, and so on. Since these are operated by interrupts, it doesn't matter what the rest of the program is doing at the time.

Figure 9.19 illustrates an example. This sets up the 16 × 16 sprite, and uses two subroutines to move. The subroutine which starts in line 1000 moves the

```
10 SCREEN2,2:SP$=""
20 FOR N%=1 TO 32
30 READ K%
40 SP$=SP$+CHR$(K%):NEXT
50 SPRITE$(1)=SP$
60 X%=128:Y%=175
70 KEY(1) ON:KEY(2) ON
80 ON KEY GOSUB 1000,2000
90 GOTO 90
100 DATA 8,4,3,67,129,65,63,15
110 DATA 27,35,198,10,18,34,66,66
120 DATA 16,32,192,192,135,137,240,22
4
130 DATA 216,196,193,160,144,136,132,
132
1000 Y%=Y%-1:PUT SPRITE 0,(X%,Y%),11,
1
1010 RETURN
2000 X%=X%-1:PUT SPRITE 0,(X%,Y%),11,
1
2010 RETURN
```

Fig. 9.19. Controlling sprites with the F keys.

sprite up the screen, the subroutine in line 2000 moves it left. When the program runs, pressing F1 will move the sprite up, and pressing F2 will move it left. Since there is no PUT SPRITE command available until you press one of these keys, the screen stays blank until an F key is pressed. This can be put to use in a 'guess where it is' type of game. Note also that line 70 has been used to specify what keys will be used in this way. Each key that you want to use like this *must* be specified. If you don't do this, the key behaves in its normal way as a function key. If you want to be able to use the same keys as function keys or for other purposes later in the program, you will have to use commands like KEY(1) OFF.

Now it's up to you. The use of sprite graphics allows much more interesting programming to be done without having to resort to controlling the machine directly using machine code. This means that with your MSX computer, you can program easily, in BASIC, games that owners of other machines have to spend weeks over. The most important things are planning and practice. This chapter has introduced you to a number of points that you don't find in the computer manual and which make the use of sprites much simpler. Try out a few ideas of your own, and before you know it, you'll be up and away, making your own programs. For a sound in your ear – read on!

Chapter Ten
Sounds Unlimited

The ability to produce sound is an essential feature of all modern computers. The sound of the MSX computer comes from the loudspeaker of the TV receiver that you use to see the display, so you have more control over the volume of this sound than is possible with a lot of other computers. In addition, the MSX computer allows you a number of different ways of creating sound effects, depending on whether you want just a reminder, a melody, or a pistol shot.

What we call sound is the result of rapid changes of the pressure of the air round our ears. Everything that generates a sound does so by altering the air pressure, and Fig. 10.1 shows how the skin of a drum does this. All other

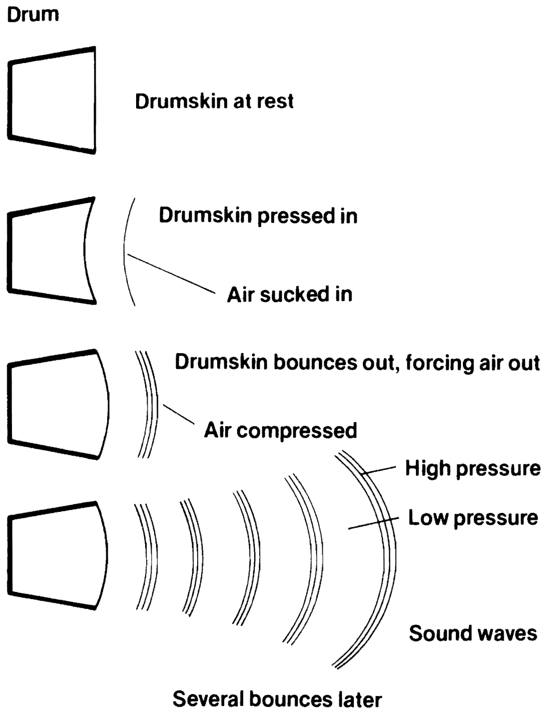


Fig. 10.1. How a vibrating drum skin creates sound waves.

musical instruments also rely on the principle of something which vibrates, and pushes the air around. Air pressure, however, is invisible, and we don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or *hertz*. A cycle of any wave is a set of changes, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Fig. 10.2. The reason that we talk about a sound *wave* is because the shape of this graph is a wave shape.

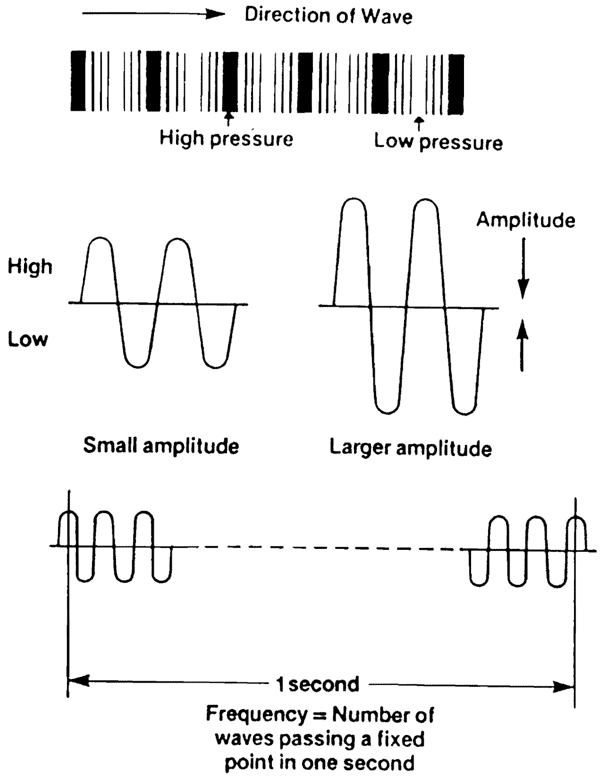


Fig. 10.2. Sound waveforms, showing how the air pressure changes with time. The number of changes per second is called the frequency. The amplitude is the maximum change of air pressure from its normal value.

The *frequency* of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20 hertz, we simply can't hear it, though it can still have disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15000 hertz. The frequency of the waves corresponds to what we sense as the *pitch* of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high-pitch treble note. Human ears are not sensitive to sounds whose frequency is above 20000 hertz (called 20 kilohertz), but many animals can hear sounds in this range.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

The MSX computer has three sound instructions, BEEP, SOUND and PLAY. In addition, there is the sound that you hear when you press a key. This 'keyclick' sound can be turned off or on with a SCREEN instruction. This can be done along with other SCREEN commands, like SCREEN 2, or SCREEN 2,1, by following the sprite number with another comma and then a 0 or a 1. A zero will turn off the keyclick; a 1 will turn it on. If you want to do this at a time when no other changes have to be made, then you can use SCREEN,,0 to turn off the clicks, and SCREEN,,1 to turn them on again. The commas are *essential* in these commands.

Of the three instructions, BEEP, SOUND and PLAY, BEEP is a simple instruction, and the notes from it have fixed pitch and amplitude. As I mentioned earlier, this amplitude is controlled by the volume control of your TV receiver, so that you can have it as loud or as soft as the TV receiver permits. The SOUND instruction is a much more complicated one, though it needs only two numbers following it. What makes it more complicated is that several SOUND instructions are needed to set up one sound, and it's designed mainly to produce sound effects that can't be produced by the other commands. We'll keep SOUND until later, and concentrate on the other two for the moment, starting with BEEP.

BEEP doesn't have to be followed by numbers, it simply causes a short sound, the same as the one which you hear along with an error message. Figure 10.3 illustrates how you might use BEEP. There is a message

```
10 CLS:PRINT"Message coming...."
20 FOR N=1 TO 1000:NEXT
30 PRINT"Hey, you......"
40 FOR N%=1 TO 10:BEEP:NEXT
```

Fig. 10.3. A program which uses BEEP to produce a sound.

appearing on the screen, and you want to make sure that the user looks at it. A long beep is produced by using a loop of ten beeps, and it makes a ringing type of noise which is quite an effective attention-getter, especially if the volume control of the TV is turned up.

You shall have music ...

The BEEP instruction is very useful for its purpose, but the MSX computer has a lot more in store for you. A lot of computers are not really suited to working with music, because they require all of the instructions to be in number form. If you read music, or can work with sheet music, this is the last thing that you want. The ideal method of programming music would be to work with the named notes of music – and this is what the MSX computer does. It might appear to be the obvious thing to do, but very few computers do it!

If you have no experience of music, however, this may seem rather puzzling to you. How do we go about writing down music? For each note, we have to specify what the note is (its pitch), how loud it must be, and for how long it is to be played. In written music, this is done by using a type of chart for the pitch, and different shapes of markings (notes) for the duration. Loudness is indicated by using letters such as *f* (loud) and *p* (soft). More than one letter can be used, so that *fff* means very loud, and *ppp* means very soft. Each sound is indicated by a note, a shape on the chart, and the shape of the note gives some information about the duration of the note. In addition to this, each piece of music will start with some advice about the speed at which the notes are to be played. One of these methods is a metronome reading. The metronome is a gadget which ticks at regular intervals, and the metronome reading for a piece of music is the number of metronome ticks per minute. A more ancient way of indicating speed is the use of italian words like *allegro* (fast), *lento* (slow) and so on. What these speed settings decide is how many unit notes will be played in a minute. The unit note is the *crochet*, so if a piece of music is marked at a metronome speed of 60 (pretty slow), then there will be 60 crochets played per minute. The durations of all the other notes are decided in comparison to this unit, the crochet. A *mimim* sounds for twice as long as the crochet; a *semibreve* sounds for twice as long as a minim, which is four times as long as the time of a crochet. The *quaver* sounds for only half the time of the crochet. A *semiquaver* sounds for only half the time of a quaver, which is a quarter of the time of a crochet. The crochets and other timed notes are indicated by the shapes of the written notes, as Fig. 10.4 shows. In addition, symbols are used to indicate silences

Symbol	Time	Name	L No.
𝅘𝅥𝅰	⅛	Demisemiquaver	L64
𝅘𝅥𝅯	¼	Semiquaver	L32
𝅘𝅥𝅮	½	Quaver	L16
♩	1	Crotchet	L8
𝅗𝅥	2	Minim	L4
𝅝	4	Semibreve	L2

Fig. 10.4. The symbols that are used in written music to indicate the time of each note, along with the MSX L number.

in the music, and these are based on the same idea of a unit duration of silence, and others which are twice, four times, half, or quarter. These rest symbols are shown in Fig. 10.5.

The pitch of a note is indicated in written music by placing it on to a kind of musical map which is called the *stave* (Fig. 10.6). Piano music uses two of these staves, each consisting of five lines and four spaces. The upper stave is the treble stave, and it is used for writing the higher notes which will be played on the piano with your right hand. The lower stave is the bass stave,

Rest Symbol	Time	R No.
𝄿	¼	R32
𝄾	½	R16
𝄽	1	R8
𝄼	2	R4
𝄻	4	R2

Fig. 10.5. The symbols for silences in written music, with MSX R numbers.

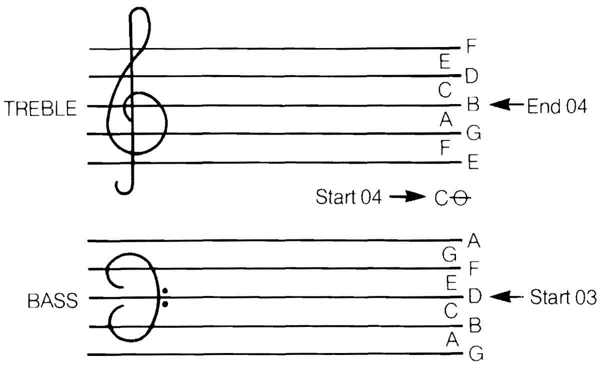


Fig. 10.6. The treble and bass staves, with the names of the notes written in.

the lower notes, played with the left hand. Instruments which do not use a keyboard will normally have music written with only one stave. In addition to this representation of notes by position on staves, we also use the letters of the alphabet from A to G to name the notes.

The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of notes, called the 'scale of C Major'. The scale starts on a note that is called

Middle C, and ends on a note that is also called C but which is the eighth note above Middle C. A group of eight notes like this is called an *octave*, so the note you end with in this scale is the C which is one octave above Middle C. Because music (in the Western hemisphere, at least) is based on this group of eight notes, we use only the first seven letters of the alphabet in naming the notes. Why 7? Well, the eighth note is the end of one octave and the start of the next, so it bears the same name. The scientific basis of all this is that if you take Middle C, and find the frequency of the sound of this note, then the C which is the next octave above Middle C has precisely double the frequency value of Middle C. The C below Middle C has half the frequency of Middle C, and so on. That's why the ancient Greeks always thought that music was a branch of mathematics.

The appearance of these keys on the piano keyboard is illustrated in Fig. 10.7. Middle C is, logically enough, at the centre of the keyboard, and we

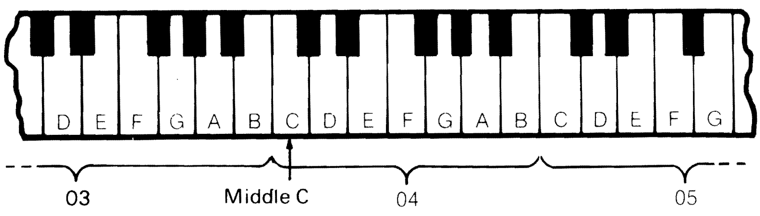


Fig. 10.7. Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and one between E and F.

move right for higher notes and left for lower notes. One of the complications of music, however, is that the frequencies of the notes of a scale are not evenly spaced out. The 'normal' full spacing is called a *tone* and the smaller spacing is called a *semitone*. Each scale will contain two semitones. On written music, Middle C appears midway between the treble and bass staves.

The key instruction for playing music on your MSX computer is the PLAY instruction. Like DRAW, PLAY has to be followed by a string name. The string then contains all the information that is needed to produce the music. The notes are specified simply by their names, as used in music. These are the letters A to G, and we also use the signs + and −. The + sign or # sign means a semitone higher than the note indicated by the letter, so that A+ or A# is a semitone above A, the note a musician would call A sharp. Similarly, A− would mean a semitone below A, or A flat. In addition to the letter names of the notes, we can use other control letters to indicate the octave, volume, length, tempo and pauses. The octave letter is O, and it has to be followed by a number whose range is 0 to 7. If you don't specify any O value, the computer will set itself to O4. O0 means the lowest range of MSX computer notes; O7 gives the highest. This means that the MSX computers can play eight octaves of notes, which is more than the range of any ordinary musical instrument. The volume control letter, V, can be followed by a number whose range is 0 to 15. This lets us make music whose volume can change during the playing of the music. As you might expect, V0 gives the lowest volume, V15 the greatest. The computer sets to V8 if you don't specify anything different. We can, of course, still set the volume control of the TV to suit our own tastes. The letter L controls the length of a note, and has to be followed by a number in the range 1 to 255. This number does not behave in the way you might expect, because the low numbers give the long notes, and the high numbers give the short notes. Figure 10.4 shows how these length numbers relate to the marked length of musical notes on sheet music. The pause or rest is a silent interval, and uses the letter R. It follows the same number scheme as note length, as shown in Fig. 10.5. If you don't specify any other values, the computer uses L4 and R4.

It's time now for some illustrations. We'll start with Fig. 10.8. This starts by defining a string A$. It consists of the notes that start at Middle C. How

```
10 A$="O4CDEFGABO5C"
20 PLAY A$
```

Fig. 10.8. The scale of C Major, by your MSX computer.

do I know? Well, Middle C on the MSX computer is the first note in octave number 4, so by starting with O4 and C, the first note that we get is C. We don't *have* to put in the O4, because this is the 'default' setting anyway, but it's a good habit to get into. The other notes have been written in sequence, but we need to put O5 before the next C. If we don't, then we'll get Middle C again instead of the C above. The scale uses the default values of volume and speed (tempo).

This is a simple scale, but it's a good piece of music to illustrate what can be done with this MSX command. Try Fig. 10.9 now, to see what we can do

```
10 A$="T120O4V1CDEFV7L50GABO5V15L1C"
20 PLAY A$
```

Fig. 10.9. Using the volume and length command letters.

with the volume V command, and the length L. The first thing that you have to know at this point is that L cannot be used in a string unless T has also been used. T means tempo, and it controls the speed of playing the string. The setting of tempo is always equal to 120 unless you change it. The range of T is a curious one, 32 to 255. The fastest tempo is obtained by using 32, the slowest by using 255. In the example, we have used the ordinary tempo, 120, but changed the volume and length of note settings. The reason for having separate tempo and length control letters is that you can get the tune sounding right by using L to select the length of notes, and then use T right at the start to set whatever tempo you like. If you want to speed things up, use a low value for T; if you want a funeral march, use a high value. You can even write the string without a T, and then add it in later by a command like:

PLAY "T100"+A$

It's time now to look at some more revelations. What's the difference between the sounds of a violin and a clarinet? You can tell which of these instruments is playing a note, even if it's the same note at the same volume. The answer is that the *shape* of the waves is different. Some method of controlling the shape of the waves, then, is an essential part of any music synthesiser. The odd thing is that very few computers have any simple way of doing this. The MSX computer does so in the shape of the S control letter. S can take a value between 0 and 15, and some of these values, along with a value for M, can make a note sound very different. The reason is that these two letters control the envelope of a note.

The word *envelope* needs some explanation. It means the pattern of a note. A musical note does not consist of just one sound wave, but of many. While a note is sounding, this volume need not be constant, though it is for the traditional electric organ type of note. For example, when you strike a piano key, the note starts very loud, and its volume then dies away as the string vibration dies away. Its envelope is therefore like the shape that is shown in Fig. 10.10 – rising very sharply, then fading away. Each musical

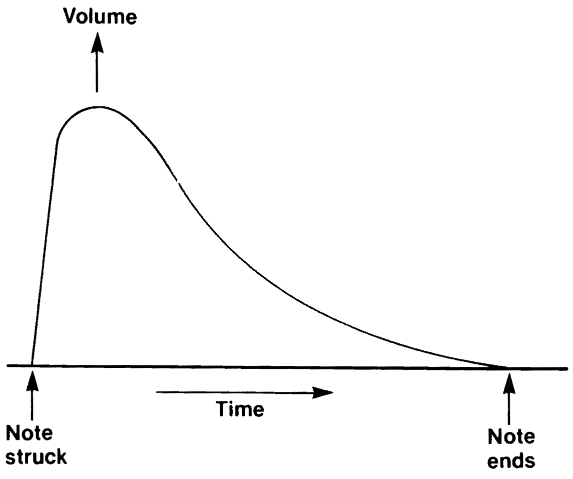


Fig. 10.10. The envelope of a piano note. This shows how the amplitude (loudness) of a single note changes during the time while it can be heard.

instrument produces its own type of envelope, and for some instruments, the effort of the player can alter the shape of the envelope. It's never easy to design your own shapes of envelopes with a computer, so the MSX computer allows you a choice of standard shapes. These are illustrated in Fig. 10.11, along with the values of S which produce them.

Used on its own, S doesn't seem to do much, but when it is combined with

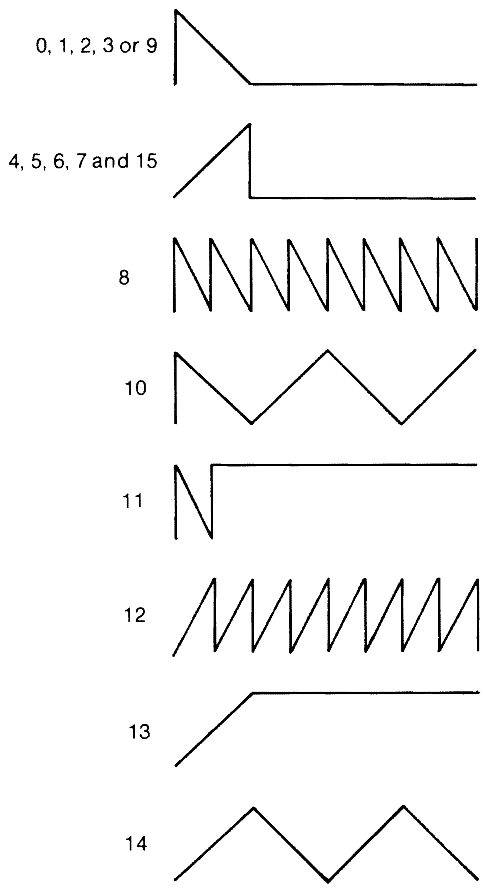


Fig. 10.11. The MSX standard envelope shapes, with the values of S that produce them.

M, it really becomes interesting. S, you see, overrides all the T and V instructions, and the combination of S and M has to be used in place of these. M can take values up to 65535, but the lower numbers from 100 to 2000 are more useful, and you have to change value by at least one hundred to hear much difference. Figure 10.12 illustrates the effect that these two have when used together. The screen prints up the values as you hear the sounds. This is a slow business, because of the time delays that are built into the program. The time delays are essential, however. The reason is that the sound generator is a little computer in its own right, and if you issue it with a PLAY command, it gets on with it independently. If you do not use a time delay in the program of Fig. 10.12, then you can find the screen displaying values of S and M well ahead of what the loudspeaker is playing. This is because the display is fast, but the music has been forced to play at a slower

```
10 A$="O4C":FOR TM!=200 TO 2000! STEP
 200
20 CLS:PRINT"M value is ";TM!
30 FOR SH%=0 TO 15
40 PRINT"S is ";SH%
50 PLAY "M"+STR$(TM!)+"S"+STR$(SH%)+A
$
60 FOR N=1 TO 1000:NEXT
70 NEXT
80 FOR N=1 TO 1000:NEXT
90 NEXT
```

Fig. 10.12. Using S and M to produce more interesting notes.

pace. This can be very useful, because it means that if you mix music with other computing actions, you will not be held up while the music plays. You will find, as you run Fig. 10.12, that several of the S values sound pretty much the same. There are in theory only eight different wave shapes in the 16 values of S, and not all of these can be easily distinguished unless you have a good ear for sound. In particular, if you use large values of M, you will not hear the effect of the 'repeater' notes that you get with S values of 8, 10, 12 and 14. You will find, in fact, that for a lot of notes, you can hear only two main types. One type is obtained by using S values of 0 to 3, or 8 to 11. The other type of note is obtained by using 4 to 7 or 12 to 15. The sound of musical notes is very much a matter of sound-it-and-see, and you always have to experiment to get exactly what you want. As you might expect from Chapter 8, there is an alternative way of incorporating variable values into a music string. Figure 10.13 shows this. The loops are both started *before* A$ is defined, and the definition of A$ starts with "M=TM!;S=SH%; which

```
10 FOR TM!=200 TO 2000! STEP 200
20 CLS:PRINT"M value is ";TM!
30 FOR SH%=0 TO 15
40 PRINT"S is ";SH%
45 A$="M=TM!;S=SH%;O4C"
50 PLAY A$
60 FOR N=1 TO 1000:NEXT
70 NEXT
```

Fig. 10.13. Putting loop counter values into S and M.

has the effect of allocating the values of the variables to the command letters M and S. Note that there *must* be a semicolon following each assignment of this type.

Figure 10.14 shows an example of a tune written using the PLAY instruction. Points to watch for here are the use of # for sharp notes, and the

```
10 CLEAR 200
20 M$="O4L4CR16CR64O3L8BR32O4CR32L4DO
3L4AR16L4G.R16L4FR16FL8EFGL2DR8L2EL4F
#GL4A.O4L4D.O3L4GO4R32CR32CR32CL8O3B.
L8AL4G"
30 PLAY "T100"+M$
```

Fig. 10.14. A tune written using PLAY. Listen to the effect of the dot following a note, and to the effect of the # sign.

use of a dot following a note letter. When you use, for example, C., then this note will play for one and a half times as long as C with no dot. The dot is used in written music in the same way, so that being able to do this with the MSX computer as well makes it all the easier to transfer written music to the form of PLAY strings. The easiest way of getting music is to use the ordinary 'organ' note to start with. You can then add the envelope commands at the start of the string. Try, for example, adding S1;M10000 at the start of the string in Fig. 10.14, then try S4;M1000 to hear the difference.

Sweet harmony

One of the many remarkable features of the MSX computers is that you can apply PLAY strings to three notes at a time, using three separate channels of sound. Three channels means that you can play up to three notes at once, and this feature allows you to have harmony. Needless to say, it requires a lot more thought, and you have to be careful to keep the three channels in step, or else what you get will certainly not be harmony. You might, of course, win a modern music prize.

Figure 10.15 shows a simple example of two-part harmony in action. The

```
10 A$="O4DR16ER16G"
20 B$="O4BR16O3GR16B"
30 PLAY A$,B$
```

Fig. 10.15. A touch of harmony, using two channels.

two strings are written so that the notes will remain in time with each other, and the PLAY instruction uses both strings, separated by a comma. If you want to play all three channels, you simply need to add another comma and another string. What is a lot less simple is writing music for this extended PLAY routine. You should, unless you have some skills in composing, work from sheet music. Music for violin and piano, or soprano voice and piano, is particularly suitable. Figure 10.16 illustrates a snatch of three-part harmony which was written from a music score. The difficulty here was to keep the notes in step, using different rest positions. You'll find that a single unit of rest, R, is enough for all but the slowest music, and often seems rather too

```
10 A$="T200O4CRL2CRCO5RC.O4RBRBL4A."
20 B$="T200O3RL2GO4CEO3GO4CEO3AO4CF"
30 C$="T200L8O3CRO2CRFRF"
40 PLAY A$,B$,C$
```

Fig. 10.16. A piece of three-part harmony.

long. Now the next thing to do is to put M4000S1 at the beginning of each of your music strings, and start experimenting. When you do this, you may find that you run into problems. If you have used M and S, and you then remove the commands by editing, you'll find that the effects do not stop! The reason is that the music computer portion of your MSX machine stores these values. To delete them, try PLAY"M0S0" (then RETURN). This produced the 'Illegal function call' error message with the machine that I was using at the time, but it sorted out the music, restoring the ordinary organ note. This can be an annoying problem if you want to run a number of different sound instructions, and it's something you will have to be careful about. It's always wise to test your final version of a PLAY routine by saving it on tape, switching off the machine, loading back and then testing. In this way, you can be certain that your sounds are not being affected by some command that you entered ten programs ago!

Finally, remember that the PLAY instruction can be used for sound effects as well. It's particularly useful for this, because you have control of many more features, like volume. Using N in a string, followed by a number between 0 and 84, will play a note corresponding to that number. Figure 10.17 illustrates the idea. If you have reset the sound commands, then you

```
10 FOR J%=0 TO 84
20 PLAY"N=J%;"
30 NEXT
40 FOR J%=0 TO 84
50 PLAY"L64T32N=J%;"
60 NEXT
```

Fig. 10.17. Using N to produce sound effects.

will get the first sequence playing slowly, and the second fast. Using N can be useful if you have a program that makes use of loops and you need something like a different note in each loop. Because of the way that you can assign a music command number to a variable value, there's scope for interesting effects here.

Sounds unlimited

PLAY is the MSX computer's gift to the musician computer owner; now let's look at what MSX can offer to games enthusiasts who want sound effects. A sound effect is a variety of noise, something that can't be written into a musical score so easily as a tune. The MSX computer uses the SOUND instruction to allow you a range of effects which go far beyond the boundaries of written music and ordinary instruments. The instructions in most of the manuals don't exactly help you with this difficult command, so I have dealt with it in a lot more detail here.

The instruction word SOUND has to be followed by two numbers, separated by a comma, but with no brackets. The first number is a *register* number. A register is a miniature memory, and it can be used to hold numbers which are usually in the range of 0 to 255 in value. The sound of the MSX computer is obtained from a separate chip, the Programmable Sound Generator (PSG) which is like a miniature computer in its own right. It uses a total of sixteen registers to store information about each sound, and the SOUND command allows us to get access directly to these registers. The PLAY instruction also makes use of these same registers, but only in fixed ways that we can't alter easily.

Each register, then, is used for controlling some aspect of the sound system, though we shall not need to make any use of registers 14 and 15. The registers are numbered 0 to 15, and the ones that we use are 0 to 13. The next step is to find what part of sound production each register controls. This is summarised in Fig. 10.18, and you will be able to make more sense of this brief reminder as we go through this chapter. For now, I'll look at the registers in turn, and explain briefly.

Register No.	Effect
0	Channel 1 frequency, fine adjustment, range 0 to 255.
1	Channel 1 coarse adjustment, range 0 to 15.
2	Channel 2 frequency, fine adjustment, range 0 to 255.
3	Channel 2 coarse adjustment, range 0 to 15.
4	Channel 3 frequency, fine adjustment, range 0 to 255.
5	Channel 3 coarse adjustment, range 0 to 15.
6	Noise predominant frequency, range 0 to 31.
7	Enable channels, see Fig. 10.23.
8	Channel 1 amplitude, range 0 to 15 (16 for envelopes).
9	Channel 2 amplitude, range 0 to 15 (16 for envelopes).
10	Channel 3 amplitude, range 0 to 15 (16 for envelopes).
11	Envelope repetition time, fine adjustment (0 to 255).
12	Envelope repetition time, coarse adjustment (0 to 255).
13	Envelope shape pattern
14	Input/output control A } Do not use!
15	Input/output control B }

Fig. 10.18. A summary of the effect of the PSG registers.

Two registers are needed to store the numbers that decide on the pitch of a note, so that six registers, numbered 0 to 5, are used for three channels. In each pair, one register is labelled 'coarse' and the other 'fine'. The 'coarse' registers (numbers 1, 3 and 5) use each number (range 0 to 15) to produce 16 notes that cover the whole range of sound. The fine registers, numbers 0,2 and 4, will accept numbers in the range 0 to 255, and adjust the note which is produced by the coarse adjustment, so that you can get a note as finely tuned as you want. You can, in fact, get 16 × 255 = 4080 separate notes by using this system. One of them must be the one you want!

Register 6 deals with noise, and can accept numbers from 0 to 63. Noise is sound which is a mixture of pitches. Very often, however, a noise has one pitch which is louder than the others. Hand-clapping, for example, has a lot of high-pitched sound, and drumbeats have a lot of low-pitched sound. The noise register allows us to pick this 'predominant frequency' for noise.

Register 7 is a selecting register. It allows us to decide how many channels of noise and/or music we pass to the loudspeaker. You can use this register to switch sounds in and out. The action is complicated, and we'll deal with it in more detail later. Registers 8,9 and 10 control the amplitude of the sound in the three music channels. The normal range of numbers here is 0 to 15, which is the range that you use in the 'V' command of a music string. If you use 16 in this register, however, you get the effect of using M in a PLAY string. This allows the volume to be controlled by an envelope. Registers 11 and 12 set the time of an envelope, and register 13 allows you the choice of shapes, using numbers 0 to 15.

SOUND in action

On to some examples. You'll probably find it useful to redefine one of your function keys, such as KEY3 to give SOUND, rather than having to type it each time. Figure 10.19 produces a note whose pitch descends. Line 30 puts the number 190 into register 7. This has the effect of turning on music channel 1, and turning off all noise channels. Figure 10.20 shows in more detail how these numbers are used. In this example, we use number 6 to select music channel 1 only, 56 to turn off all noise channels, and add 128 to these numbers to get 190. Line 40 then puts the number 15 into register 8.

```
10 CLS
20 FOR N%=1 TO 255
30 SOUND 7,190
40 SOUND 8,15
50 SOUND 0,N%
60 NEXT
70 SOUND 8,0
```

Fig. 10.19. Programming a note of descending pitch.

Register 7

Music channel 1 only	6
No noise	56
Add 192	192
Total	254

The figure of 15 in register 8 gives full volume on music channel 1.

Fig. 10.20. The details of how the numbers for a sound effect are selected for Fig. 10.19.

This sets the volume of sound in channel 1 to its maximum value. The control of the pitch of the note is then carried out by using register 0. This is the 'fine' control for channel 1. If you change the number in register 1, you will get a different range of notes. By using N as a counter, and placing N in the register, we get a note whose frequency changes. After the loop has finished, we need line 70 to shut off the sound. Without line 70, the sound keeps going until you press the CTRL/STOP keys.

Now for a bit of amusement. The program in Fig. 10.21 uses two channels. This is achieved in line 30 by placing the 'full-volume' number of

```
10 CLS
20 SOUND 7,188
30 SOUND 8,15:SOUND 9,15
40 FOR N%=1 TO 255
50 SOUND 0,N%
60 SOUND 2,256-N%
70 NEXT
80 SOUND 7,191
```

Fig. 10.21. Controlling two channels with the SOUND command.

15 into registers 8 and 9, which control channels 1 and 2 respectively. The pitch numbers are placed in lines 50 and 60, using the loop number N% for register 0 (channel 1) and 256–N% for register 2 (channel 2). The number N% will create a note whose pitch decreases as N% increases, and 256–N% will produce a note whose pitch increases as N% increases.

Figure 10.22 takes us a few stages further along the road. Line 20 enables all three of the sound channels, and line 30 sets maximum volume in each channel. The main loop that starts in line 40 then gives the same combination as is used in Fig. 10.21, but this time we have added a different sound in the third channel, by means of another loop. Since the other two notes keep playing while this is altering, you hear the effect of all three until blast-off is achieved!

```
10 CLS
20 SOUND 7,184
30 SOUND 8,15:SOUND 9,15:SOUND 10,15
40 FOR N%=1 TO 255
50 SOUND 0,N%
60 SOUND 2,256-N%
70 FOR J%=1 TO 20
80 SOUND 4,J%:NEXT
90 NEXT
100 SOUND 7,191
110 PRINT"BLASTOFF!"
```

Fig. 10.22. Three-channel SOUND.

Some of the most impressive sound effects that the MSX computer can produce require the use of the noise generator. Noise is a mixture of frequencies, unlike a musical note which always has one clear 'fundamental' frequency. Noise may nevertheless have a 'predominant' frequency, meaning that most of the noise frequencies are centred around this frequency instead of being spread evenly over all the range of frequencies. The noise generation of the MSX computer depends on the use of registers 6 and 7.

We'll start with register 7 because it's the use of this register that allows noise signals to be sent to the three channels. Your choices in this matter are made by the number that you put in following SOUND 7, and Fig. 10.23 lists these numbers, and how values can be added to mix the effects. The number that you put into register 7 is, in fact, the sum of 128 plus separate numbers for the tone and for the noise channels.

Channels activated	Tone code	Noise code
A, B and C	0	0
B and C only	1	8
A and C only	2	16
C only	3	24
A and B only	4	32
B only	5	40
A only	6	48
None	7	56

Add 128 to the sum of the number(s) used.

Example: Tone on channels A and B, noise on channels B and C codes are 4 and 8, which add to 12, then add 128 to get 140. This, then, is the number that is placed in register 7.

Fig. 10.23. How numbers are used in Register 7 of the PSG.

Register 6, by contrast, uses numbers that can range from 0 to 31. This is the register that causes the sound to have a predominant frequency. As an illustration of the effect of predominant frequency, try the program in Fig. 10.24, which produces a rather impressive 'surf on the shore' type of noise.

```
10 CLS
20 SOUND 7,183
30 SOUND 8,15
40 FOR X%=1 TO 20
50 FOR N%=0 TO 31
60 SOUND 6,N%
70 FOR J=1 TO 50:NEXT
80 NEXT
90 NEXT
100 SOUND 8,0
```

Fig. 10.24. The surf on the shore program.

The figure of 183 that is put into register 7 is made up of the usual 128, plus 7 for 'no tones' and 48 for 'noise channel 1 only'. The loudness of channel 1 is put to full amplitude by line 30. We select 20 waves in line 40, and then the loop in lines 50 and 60 carry out the wave sound. The noise predominant frequency starts high, with N%=0, causing a hissing sound, and ends up with the booming noise that is caused when N%=31. We can now use these noises as a basis for more useful sound effects.

Opening the envelopes

We've mentioned the principle of envelopes earlier, and it's time now to see how the MSX computer can make use of such envelopes with the SOUND command. Figure 10.11 showed the envelope shapes from which you can choose, and the numbers that you use for the SOUND command are the same. Figure 10.25 demonstrates the effects of the envelopes. The important

```
10 CLS
20 SOUND 7,190
30 SOUND 0,150
40 FOR N%=0 TO 15
50 SOUND 13,N%
60 PRINT"Envelope No. ";N%
70 SOUND 8,16:SOUND 12,10
80 FOR J=1 TO 1500:NEXT
90 NEXT
```

Fig. 10.25. Using the standard envelopes with the SOUND command.

command is in line 70, and is SOUND8,16. When 16 (or any number between 16 and 31) is used in register 8, its effect is to allow the envelope-generating part of the sound system to take control of amplitude. The sound will no longer have a fixed amplitude, but will take values that depend on whatever envelope has been chosen. The other lines are more conventional. Line 20 enables music on to channel 1, and line 30 puts a note number into the channel 1 register. Register 13 is then used to select envelopes. As you can also see from Fig. 10.25, several numbers produce the same envelopes. You can listen to the effects of each envelope, and compare the sounds that you hear with the appearance of the shapes in Fig. 10.11. The delay loop in line 80 gives plenty of time for one sound to be completed before the next one starts. An important point here is that you must have *something* put into register 12 (see line 70). This one controls the envelope period, and if you do not place a number in here, all that you will hear will be clicks – unless the register has been filled by a previous command.

We can produce some rather useful tinkling notes with envelope 1, as Fig. 10.26 illustrates. In this example, line 20 sets the envelope period at a

```
10 CLS
20 SOUND 12,10
30 SOUND 7,190
40 SOUND 8,16
50 FOR N%=255 TO 1 STEP -10
60 SOUND 13,1
70 SOUND 0,N%
80 FOR J=1 TO 200:NEXT
90 NEXT
```

Fig. 10.26. A program which demonstrates the effect of envelope 1.

number which gives a short note. This is a quantity which can be experimented with, but if you make the period number bigger, then you will need a longer delay time in line 80 also. Line 30 enables channel 1, and line 40 allows the envelope generator to control the amplitude. The loop then selects envelope 1 on each pass through the loop. Line 70 puts different note numbers into the pitch register for channel 1, and the result is – well, listen for yourself!

By way of contrast, Fig. 10.27 demonstrates what happens when we use noise along with an envelope control. The noise is selected by line 20, and the rest of the instructions should be reasonably familiar to you by now – note

```
10 CLS
20 SOUND 7,183
30 SOUND 8,16
40 SOUND 6,8
50 SOUND 13,8
60 FOR N=1 TO 5000:NEXT
70 SOUND 8,0
```

Fig. 10.27. A drum-beat noise program.

the use of SOUND 8,0 to turn off the sound at the end of the program. The drumming continues for the duration of the delay loop – you don't need to have the SOUND instructions *inside* a loop. Finally, try the program in Fig. 10.28, and hear what happens when we slow the drummer down a bit. You should be able to analyse what's happening here for yourself! How about speeding it up a bit, and making it sound like a steam loco in full cry?

```
10 CLS
20 SOUND 7,183
30 SOUND 8,16
40 SOUND 6,15
50 SOUND 13,8
60 SOUND 12,30
70 FOR N=1 TO 5000:NEXT
80 SOUND 8,0
```

Fig. 10.28. Converting the drum-beats into hammer blows!

Chapter Eleven
Cassette Data Filing

Many small computers make no provision at all for recording data, as distinct from recording programs, on cassette. Because of this, the use of such a system for data recording will probably be quite new to owners of a MSX computer, even if they have used a computer previously. This chapter, then, is devoted to the use of the cassette recorder system for storing data. Your manual will have dealt with the use of the cassette system in connection with storing and loading programs, and Appendix A deals with adjustment of the cassette recorder for best results. You should make sure that you fully understand the use of the recorder for program storage before you continue with this chapter. You should also make sure that you have read the references to cassette recorders in Appendices A and B.

The most puzzling part of work on data recording is the number of new names and ideas that you encounter. The new names that cause difficulty to the MSX computer owner are 'devices, streams and buffers'. Once you grasp what is meant by these words, and how they are applied, you will find cassette system operation much more interesting, and you will also be able to do much more with your MSX computer. Let's start, then, by explaining these words. A *device* is something that puts out or receives data. Your keyboard is a device, because each time you touch a key, a set of electrical signals is sent to the computer. The screen is another device, because every time the computer sends a set of electrical signals to the TV set, you will see something appear on the screen. The keyboard is a transmitting device, because it sends signals. The screen (or the TV) is a receiving device, because it accepts signals.

Some devices can perform both operations. The cassette system is also a device which can be used in both directions. The disk system is a similar type of device. MSX computers identify devices by abbreviations, using CAS: for the cassette recorder, CRT: for the text screen, GRP: for the graphics screen, and LPT: for the printer.

We have talked of electrical signals passing from one device to another, and this is what actually happens. It's a lot more useful to think of what these signals represent. Each set of signals represents a unit of data called a *byte*, and data is the stuff that computers are designed to deal with. One byte is the amount of memory that is needed to store one character in ASCII code, or one command word in BASIC. Data may be numbers or names; it's anything that the computer has to work with. If you have a program that arranges the names of your friends in order of birthdays, then that program needs data. The data in this example is the set of names and birth dates. If you have a program that shows cookery recipes and shopping lists, then the data is the instructions, the names of the foodstuffs, and the quantities. Every computer that is designed to be used for anything more than the simplest games must be able to save and load data of this type separately from the program that generates it or uses it.

There's a lot to be gained from this approach. The memory of the MSX computer is used for quite a lot of purposes over which you have no control. A very long program which gathered data and then made use of it might not fit into your computer. It's a lot more sensible to have a short program which gathers the data, using INPUT lines, and which records the data as it is gathered. The data is then safe if anything should happen (like a momentary failure of the power supply) that scrambles the memory of the MSX computer. Another program can then make use of this same data. By keeping the two programs and the data separate, you can deal with a lot more information than would be possible if you had to have the whole lot in the memory at one time.

What has all this to do with buffers, streams and devices? Well, *devices* are the parts of the computer which give out or receive data, and *streams* are the paths which carry the data. Just think of what happens when you use your MSX computer. When you press a key, something appears on the screen. The keyboard is one device, the screen is another, and there is a stream which links them. This is just a fancy way of saying that there is a path for data signals from the keyboard to the screen. The important point, however, is that these paths or streams can be controlled. Controlling them means that we can change the paths, breaking some and making others, as we please. It wouldn't be sensible to break some of the paths, of course, except for special purposes. You normally want to see on the screen the words that you type on the keyboard. If you were typing a special password, however, and you didn't want anyone who was watching the screen to see it, it would make sense to break the stream that connects the keyboard to the screen, and we have already done this by using the INPUT$ command.

As you might expect by now, there are some streams which are connected to devices from the moment you switch on. It's obvious, for example, that there's a connection between the keyboard and the screen. MSX computers, however, allow you to make more connections, using the device abbreviations of CAS:, CRT:, GRP: and LPT: as listed above. In this chapter, we are concerned with the cassette recorder, CAS:, and we shall make no further use of the other device abbreviations. We can also make use of numbered streams which connect to the cassette recorder. We can use up to 15 of these numbered streams, numbered 1 to 15, *not* as you might expect 0 to 14. For each stream, the computer has to set aside a portion of memory to use for storing data that is to be written (recorded) or read (replayed). This portion of memory is called a *buffer*. In order to use the cassette recorder for data storage you have to know two things. One is how to select a buffer, the other is how to link it to the cassette recorder.

The way that you select a buffer and link the cassette recorder is by using the OPEN command. You have already seen this in action in Chapter 7, being used to place text on the graphics screen. For cassette use, the OPEN command must contain rather more information. You must specify the device name, which will be CAS: for cassette recording. You then need a filename, so that you can identify the collection of data (the file) when you want to replay it. You may, after all, have several different files on one cassette. You need to specify whether you want to use the stream for output or for input, and finally you need to select a stream number in the range 1 to 15. For example, you might use:

```
OPEN"CAS:DATA"FOR OUTPUT AS #1
```

When this runs, it will link the cassette recorder to the computer ready for data to be recorded. This means an output from the computer, so that OUTPUT must be used in the command. The filename is DATA, and this follows the CAS: which must be used. The filename must be of six letters or less (*not* including CAS:), and must *not* contain a colon : or the numbers 0 or 255. The computer uses the colon to recognise device words like CAS:, and the numbers 0 and 255 are also used as terminators. The #1 is the stream number and our number in this example is 1. (The hashmark # is the American way of writing what we write as No., the number of an item.) The command therefore prepares stream 1 for an output to the recorder, using the filename of DATA. This is not simply a preparation; when this statement runs, the cassette motor will operate, and the filename will be recorded on the tape. To open a file for input with this filename, we would need something like:

```
OPEN"CAS:DATA" FOR INPUT AS #2
```

using stream #2 this time. By specifying different stream numbers like this, we can switch from writing to reading very quickly and easily. When this statement runs, also, because of the use of the filename, only a file of the correct name will be selected, and the cassette system *will be operated* so as to find this filename on the tape.

Data filing techniques

What is a file?

The word *file* occurs many times in the course of this book. A file can mean any collection of characters which belong together. The characters of a BASIC program constitute a file, for example, because the program will not run if characters are missing. A set of names and addresses in ASCII code is a file, because they form one group of information, such as our friends, or our suppliers, or debtors. A set of bytes of machine code is a file, and a collection of the numbers that are used by a financial program is a file. In this chapter, though, I'll take file to mean a collection of information which we can record on a cassette, and which is separate from a program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the action of the program, and it preserves these amounts for the next time that you use the program. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings. This information is a file and, at some stage in the program, you would have to record this file. Why? Because if a program like this is going to be really useful, there will not be enough space even in the memory of your MSX computer to hold all of the information at one time. In addition, you wouldn't want to have to change the program each time you wanted to add items to the list. This is the topic that we're dealing with in this chapter – *recording* the information that a program uses. The shorter word is *filing* the information.

You can't discuss filing without coming across some words which are always used in connection with filing. The most important of these words are *record* and *field* (Fig. 11.1). A record is a set of facts about one item in the file. For example, if you have a file about LNER steam locomotives, one of your records might be used for each locomotive type. Within that record,

	FRIENDS FILE
RECORD 1	
FIELD 1	Name 1
FIELD 2	Address 1
FIELD 3	Phone No. 1
FIELD 4	Birthday 1
RECORD 2	
FIELD 1	Name 2
FIELD 2	Address 2
FIELD 3	Phone No. 2
FIELD 4	Birthday 2
RECORD 3	
etc.	

Fig. 11.1. The meaning of record and field. Here, each record is for a particular friend, with different fields for name, address, telephone number and birthday.

you might have designer's name, firebox area, working steam pressure, tractive force ... and anything else that's relevant. Each of these items is a *field*, an item of the group that makes up a record. Your record might, for example, be the SCOTT class 4-4-0 locomotives. Every different bit of information about the SCOTT class is a field, the whole set of fields is a record, and the SCOTT class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on. Take another example, the file British Motor-bikes. In this file, BSA is one record, AJS is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, suspension, top speed, acceleration ... and whatever else you want to take note of. Filing is fun – if you like to arrange things in the right order.

Cassette system filing

In this book, because we are dealing with the MSX computer cassette system, we'll ignore filing methods that are based on DATA lines in a BASIC program. This is because cassette data filing keeps the data separate from the program, and is therefore much more useful. If it's all familiar to you, please bear with me until I come to something that you haven't met before. To start with, there are two types of files, only one of which we can use with a cassette system. These are serial files and random access files. The difference is a simple but important one. A serial (or sequential) file records all the information in order on a cassette system, just as it would be placed on a cassette. If you want to get at one item, you have to read all of the items into the computer, and then select. There is no simple way in which you can command the system to read just one record or one field. A random access file does what its name suggests – it allows you to get from the recorded data one selected record or field without reading every other one from the start of the file. The difference between serial filing on tape and random access filing on disk is illustrated in Fig. 11.2. Random access filing is something which requires the use of a disk system, but we can design programs which achieve something like the same effect by using serial files on the cassette system. We'll start, then, by looking at serial files, which are also the type of files that we record on a cassette.

Creating a file

When you are dealing with something new, it's always a good idea to start at the beginning and keep things simple at first. We'll start the idea of filing with the way that we make connections to the cassette system. Fig. 11.3 shows a very simple example of how one item of data, the value of a variable called A%, can be recorded on the cassette system, whose stream number

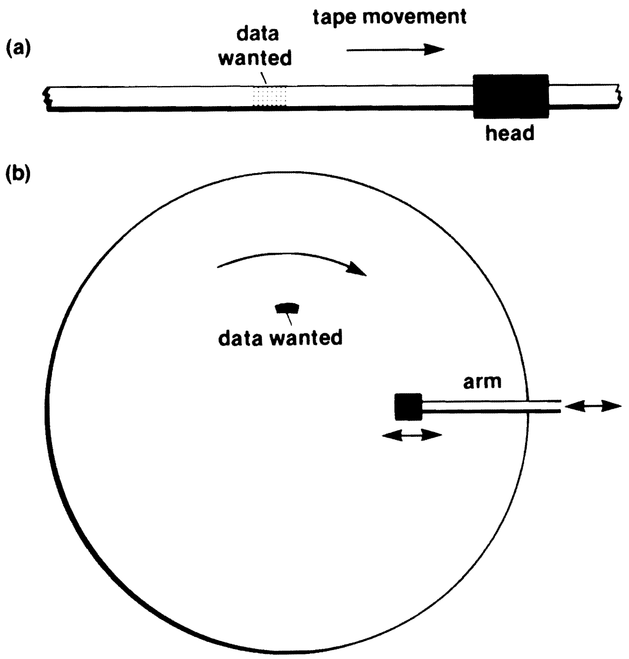


Fig. 11.2. (a) Serial filing on cassette and (b) random access filing on disk.

```
10 OPEN"CAS:TEST" FOR OUTPUT AS #1
20 A%=5
30 PRINT#1,A%
40 CLOSE
```

Fig. 11.3. Recording the value of a single variable. It's the value that's recorded, not the variable name.

has been chosen to be 1. Because the steps are so important, we'll look at each of them in close detail. Starting with line 10, then, we open a file. This requires the command word OPEN, with "CAS:TEXT" used as the device and filename. The file is an OUTPUT file, recording on to the cassette, and its stream number is #1. You must be careful about how you use the recorder here, because you need to select a cassette, and a position on the cassette, which has nothing else recorded on it. There is nothing to prevent you from 'wiping' a file by accidentally recording over it another one which has the same name, or a different name. Disk systems can protect you against that sort of error, but when you use a cassette for data storage, you only have the reading of the tape counter to help you. The cassette recorder also needs to be switched to record *whenever this program runs.*

The next step is to assign a value to the variable, A%, in line 20. Line 30 is the important one now. The instruction PRINT#1,A% means send out the value of A% over stream 1. Stream 1, however, has been connected to the cassette system by the OPEN statement, so that this line 30 will 'print' the value of A% on to the cassette. Press the RECORD and PLAY keys of the recorder, and run the program. You do *not* see the usual messages that you get when you use SAVE, and the only indication that you get when this runs is the sound of the cassette motor, and the O.k. prompt when the program is ended. The value of A% (but *not* the variable name of A%) will be filed under the name of TEST so that it can be easily found again. Line 30 causes the recording to be made, but it's not quite so straightforward as it might seem. The cassette system records groups of characters, and the operating system is arranged so that the computer will gather up data in the memory until it has enough. This part of memory is called a *buffer*, and when you open a stream, you also automatically allocate a buffer for that stream. The data that is to be recorded is shifted into the buffer, and is then recorded. If there is more data than the buffer can cope with, then the buffer will have to be filled and emptied more than once. At the end of such a process, you have to make sure that the buffer is cleared. This is done in line 40 by the CLOSE command. Now what happens when this runs? As far as you are concerned, it's just that the cassette system starts, spins for rather a long time, then stops.

Now we have to prove that this data was actually recorded, and show that we can recover it. Take a look now at the listing in Fig. 11.4. Line 20 in this listing uses OPEN"CAS:TEST" FOR INPUT AS #2. We already have a file

```
10 MAXFILES=2
20 OPEN"CAS:TEST"FOR INPUT AS #2
30 INPUT #2,X%
40 PRINT"X% IS ";X%
50 CLOSE
```

Fig. 11.4. Recovering the variable from the tape.

on our cassette, and we want to read it, not to create a new file. The stream number this time is #2, and we have to specify "TEST", the name of the file. It won't work, though, without line 10. This isn't because we are using INPUT in place of OUTPUT, it's because we have used a stream number greater than 1. The machine keeps one buffer ready for use, and this can be allocated to #1. If you want to use numbers like #2, #3 and so on, you have to prepare the memory. This is done by the MAXFILES command. By using MAXFILES=2, you make the computer provide memory space for files which use stream numbers up to 2. This MAXFILES command *must* be carried out right at the start of a program, because it has drastic effects. Because it prepares memory, it will erase anything that is already in memory, apart from the program itself. If you have dimensioned an array, used DEFINT to declare integer variables, or assigned any variables, MAXFILES will wipe it all out. The only really safe place for it is in the first line of the program!

Having done this and opened the file, meaning that a buffer will now be allocated for use with signals from the cassette system, the system finds the filename on the tape, if it exists. You must, at this stage, press the PLAY key on the recorder. If the filename cannot be found, the system will keep on trying as long as there is tape to read! A disk system does this type of thing much better, because it can find at once if the filename is on the disk. Having found the file, we can then read the data. Line 30 does so, using INPUT #2,X%. INPUT by itself always refers to the keyboard, but when we place the #2 after INPUT, it will cause the input to come from the specified stream, #2, which means the cassette system because of the use of OPEN. Because we have specified "TEST" in the OPEN statement, what comes from the cassette system will only be whatever is in the file TEST. Line 40 prints what is read in, and line 50 closes the stream down again. By using CLOSE, *all* streams are closed. You could close a specified stream by using CLOSE #2, for example. It all looks reasonably simple and straightforward, but take a close look at these two little pieces of programming, because they contain a lot that you will need to get to grips with in the course of data filing. Notice, for example, that we can assign what is read to any variable name that we like. We used A% when recording, but X% when replaying. As far as the computer is concerned, reading a file from cassette is just another INPUT step like reading from the keyboard. Notice also that even the simplest programs of this kind need some messages on the screen to remind you that you have to press the correct keys on the tape recorder. This is another important difference between using the tape recorder and using a disk system.

Now try something more ambitious with the creation of a file of numbers. Figure 11.5 shows a program which generates a file of numbers – the even numbers from 0 to 50 – then records these numbers on the cassette system and also prints them on the screen. There are only six lines to this program,

```
10 OPEN"CAS:EVENS" FOR OUTPUT AS #1
20 CLS:FOR N%=0 TO 50 STEP 2
30 PRINT #1,N%
40 PRINT N%;"  ";
50 NEXT
60 CLOSE
```

Fig. 11.5. Creating a file of numbers and recording the values.

but three of them contain these important commands that you need to understand. We'll start, reasonably enough, with line 10. This is one of these OPEN commands which connects a buffer to a stream. The stream is #1, so that we don't need to use MAXFILES, and the filename is EVENS. The next thing is to create a file, and in this case, it's being done by a loop which starts in line 20. This allocates variable N% as each of the even numbers in turn, with the NEXT in line 50. How do we place the numbers into the buffer? Line 30 does this, using PRINT #1,N%, and since there is a string of numbers to be recorded, the buffer accepts the numbers *before* the cassette recording begins. The cassette system is nothing like as fast as the computer. The FOR...NEXT loop in lines 20 to 50 could easily be completed before the motor of the cassette system could be started! Each time line 30 runs, the value of N% is stored temporarily in the buffer. Buffer is a good name, because its action is to connect the computer and the cassette system so that they work smoothly together. In this simple example, all the numbers that are generated by the action of the loop are simply passed into the buffer. Nothing is recorded in this time, the cassette system motor has stopped after recording the filename, and before recording the data. The recording of data *then* takes place when all of the numbers have been assembled. CLOSE means close down all streams, and when an output stream is closed, part of the action is to empty any buffer which is part of that stream. In addition, an end-of-file marker is recorded, so that the system can identify the end of a file even when several blocks have been recorded. That's it! If you now press RECORD and PLAY, and RUN this program, you'll hear the motor run to record the filename, then see the numbers appear on the screen, showing that the whole loop is being run. You will then hear the motor run to record the data only after the whole loop has ended. Once again, if you have forgotten to press RECORD and PLAY on the recorder, the computer goes through the motions, but nothing can be recorded!

This program has shown you the buffer in action, and the size of the buffer is large enough for a lot of data. You can try it for yourself by altering the program so that it looks like Fig. 11.6. Lines 10 to 30 provide a suitable

```
10 CLS:PRINT"Please press REC and PLA
Y keys NOW."
20 PRINT"Press SPACEBAR to start."
30 IF INKEY$<>" "THEN 30
40 OPEN"CAS:MOREVN"FOR OUTPUT AS #1
50 CLS:FOR N%=0 TO 500 STEP 2
60 PRINT#1,N%
70 PRINT N%;"  ";
80 NEXT
90 CLOSE
100 PRINT: PRINT"End of recording- pl
ease press STOP":PRINT"key of recorde
r."
```

Fig. 11.6. A much longer number file, to display buffer action.

message and a pause to give you time to prepare the recorder. This time, the number of bytes of data will need several thousand bytes, and when you RUN the program, you will find that the numbers appear on the screen until 84 is reached. At this point, the cassette motor starts, and a block of data is recorded. The numbers start running again until 158, when the cassette records another block, and the same happens at 230, and every 72 numbers after that. What is happening is that the buffer is being filled because of the loop, and is being emptied by the cassette system.

These short programs have put data into a file, but so far, you have had to trust me that there is actually something on the tape. Figure 11.7 shows how

```
10 CLS:PRINT"Press PLAY key, then SPA
CEBAR."
20 IF INKEY$<>" "THEN 20
30 OPEN"CAS:EVENS"FOR INPUT AS #1
40 FOR N%=0 TO 50 STEP 2
50 INPUT#1,A%
60 PRINTA%;" ";
70 NEXT
80 CLOSE
90 PRINT:PRINT"Press STOP key of reco
rder."
```

Fig. 11.7. Reading the EVENS file. You need to have a PRESS SPACEBAR step early on to give you time to prepare the cassette recorder.

this can be done for the EVENS file. The program starts as usual with the message about pressing the PLAY key of the recorder. This is because when the OPEN command is carried out, the machine must find the filename of EVENS before anything else can be done. You therefore need to have your 'PRESS PLAY' message right at the beginning of the program, so that OPEN can do its work. As usual, the program starts in line 30 with opening the file, using OPEN, with the filename of EVENS. We read the file with INPUT#1, in a loop to read and display the data. When this runs, then, you will hear the cassette motor run, the machine will locate the file, and read it, and you will see the numbers appear on the screen. Note that you get no message which tells you that the computer has found the correct file. All that you have to guide you is a clicking sound inside the machine when the correctly named file is found.

Now let's try the longer file, MOREVN, in Fig. 11.8. This time, we'll arrange a more tidy screen display by interrupting the display process. Line 80 performs the interruption. The condition is IF N%/40=INT(N%/40). This means the condition when N% divides evenly by 40. If N%/40 has a remainder, then it can't be equal to INT(N%/40), which is the whole number part of N%/40 only. Each time N% has a value that divides evenly by 40, then, the second part of line 80 causes a delay, and at the end of the delay, the screen is cleared before the main loop continues. By using N%/40, you place 20 numbers on the screen because N% increments in twos, remember. Could you, perhaps, do this in a different way by using MOD? Now when you run this one, you will hear the cassette motor start, find the file, and read it. The number 0 then appears under the text on the screen, and then the main loop displays the numbers in groups of twenty. The cassette motor will start and

```
10 CLS:PRINT"Press PLAY key on record
er-"
20 PRINT"-then spacebar to start."
30 IF INKEY$<>" "THEN 30
40 OPEN"CAS:MOREVN"FOR INPUT AS #1
50 FOR N%=0 TO 500 STEP 2
60 INPUT #1,K%
70 PRINTTAB(12)K%
80 IF N%/40=INT(N%/40)THEN FOR J=1 TO
 2000:NEXT:CLS
90 NEXT:CLOSE
100 PRINT"Press STOP key of recorder.
"
```

Fig. 11.8. Reading the MOREVN file, with a screen display that gives you time to see the numbers.

stop at intervals as required to fill the buffer. You will hear this stop and start action going on all the time that data is being read back from the MOREVN file. The process is a slow one, and it's better if data filing is carried out at a faster speed. You can't alter the speed at which the tape moves, but it *is* possible to read and write more bytes per second. This is done by making use of another variation on the SCREEN command. This needs to be done *only for recording*. When the machine reads a tape, it will set itself for the correct rate of reading data. The command for using the higher speed is SCREEN,,,2 assuming that you are not using any of the other SCREEN options. If you are, then there will be other items between the commas. To reset to normal speed, use SCREEN,,,1. From now on, we'll use the faster speed.

More serial filing

Suppose that what we want to record is not a set of numbers that has been generated by a program, but a set of names that you have typed. As far as the cassette data system is concerned, this is just another set of data, and it's dealt with in exactly the same way. Each time you press RETURN at an INPUT step, the data is stored in a buffer, and it stays there until the buffer is full, or until the entry is complete and the file is closed. Once again, you can see the importance of using a buffer – you wouldn't expect the cassette data system to record each letter as you typed it, would you?

Figure 11.9 shows a short program of this type. Normally, if you were gathering information like this, you would store the names of an array. This, as you probably know, introduces complications like having to dimension the array. Unless you want to look at a previous entry at some time when you were entering names, however, you don't need to use an array for recording a set of names. It can be a different matter when you play back, but that's something that we'll look at shortly.

```
10 SCREEN,,,2
20 CLS:PRINT:PRINTTAB(16)"NAMES"
30 PRINT:PRINT"This program stores a
file of names":PRINT"for you on the c
assette. Make sure":PRINT"that you ha
ve a cassette ready."
40 PRINT:PRINT"Input X to end entry."
50 PRINT:PRINT"Press REC and PLAY on
the recorder":PRINT"and then the spac
ebar when you":PRINT"are ready."
60 IF INKEY$<>" "THEN 60
70 PRINT"Please wait.................
."
80 OPEN"CAS:NAMES"FOR OUTPUT AS #1
90 N%=N%+1:PRINT"Name";N%;"  ":INPUT N
M$
100 PRINT#1,NM$
110 IF NM$<>"X"THEN 90
120 CLOSE
130 PRINT"Press STOP key of recorder
now."
140 SCREEN,,,1:END
```

Fig. 11.9. Filing names. The faster cassette filing speed has been selected to make the program more efficient. The names don't have to be put into an array.

Taking the program in more detail now, line 10 selects SCREEN,,,2 for the faster cassette writing speed. This does *not*, remember, mean that the cassette tape moves faster, just that the data is recorded on to it at a higher rate. We shall cancel this command at the end of the program, because if we don't it will remain in force. You might want to SAVE another item at the slower rate, and this would not be possible after the higher speed has been selected. Lines 20 to 50 print the usual messages and instructions, and line 60 is the INKEY$ line. Line 70 prints another message, because it can be confusing when you press the spacebar and nothing seems to be happening. Line 80 then opens the file, so that the filename is recorded. Only when this has been done will the first number and the request for a name appear.

The main loop then starts in line 90, and the idea is to input and record each name until an X is typed. If you wanted to expand this into a more realistic name file, you would probably want more detailed instructions. Line 90 accepts the name that you type (no commas permitted with an INPUT step, remember). The name is then recorded in line 100. If X has not been entered, then line 110 returns for the next word. If you type names fast and continually, you will find that the cassette motor spins every now and again, emptying the buffer. You can't enter data while this is going on.

More on reading

By this time, you have a few files of both numbers and names stored on your cassette data system, and it's time to pay rather more attention to the methods that are used for reading files and using the information from them. Suppose, for example, that the numbers which we recorded in the file EVENS had been placed on the file by an accounts program. They might, for example, be the daily takings of a small shop. One thing that we might want to do with the numbers, then, would be to read them from the file and add them, showing only the total. This is a conventional and straightforward piece of programming, and one for which you will probably find a large number of uses. Figure 11.10 shows what is needed. It starts in line 10

```
10 CLS
20 PRINTTAB(15)"TOTALS"
30 PRINT:PRINT"Press PLAY key on reco
rder when the":PRINT"cassette is read
y."
40 PRINT"Press SPACEBAR to start read
-in."
50 IF INKEY$<>" "THEN 50
60 OPEN"CAS:EVENS"FOR INPUT AS #1
70 TT=0
80 FOR N%=1 TO 26
90 INPUT #1,J%
100 TT=TT+J%
110 NEXT
120 PRINT:PRINT"Total is ";TT
130 CLOSE:END
```

Fig. 11.10. Reading back and totalling numbers from the EVENS file, using a FOR...NEXT loop.

by clearing the screen and then line 20 prints the title, followed by some instructions in lines 30 and 40. Line 50 causes the machine to wait for the spacebar to be pressed, and the real work starts in line 60, which opens the EVENS file. Now when we recorded the file EVENS, we selected all the even numbers from 0 to 50, which is a total of 26 numbers. To read the same set back, then, line 80 uses a FOR...NEXT loop with 26 passes. The number TT has been set to zero in line 70. Line 90 then inputs each number from the file, giving it the variable name of J%, and line 100 adds this number to the total. At the end of the loop, line 120 prints the value of the total and the file is closed in line 130.

Suppose that you didn't know how many numbers were recorded? This makes the use of a FOR...NEXT loop impossible, because you wouldn't know what number of passes to use. As it happens, we can cope with this quite easily. The MSX computer filing system puts an end-of-file code at the end of the last block of data that it records. Now this end-of-file code can be detected by using the word EOF, which has to be followed by the file number, in brackets. If EOF(1)=0, the end is not yet nigh. If EOF(1)=−1, then you have reached the end of the file. Note that the file number is 1, because this is the number of the file that has been opened for input. There *must be no hashmark* with this number – a line which includes EOF(#1) will be rejected as a 'Syntax error'. Our EVENS file can therefore be much more conveniently written as in Fig. 11.11. This time, we use a GOTO loop in

```
10 CLS
20 PRINTTAB(15)"TOTALS"
30 PRINT:PRINT"Press PLAY key on reco
rder when the":PRINT"cassette is read
y."
40 PRINT"Press SPACEBAR to start read
-in."
50 IF INKEY$<>" "THEN 50
60 OPEN"CAS:EVENS"FOR INPUT AS #1
70 TT=0
80 INPUT #1,J%
90 TT=TT+J%
100 IF EOF(1)=0 THEN 80
110 PRINT:PRINT"Total is ";TT
120 CLOSE:END
```

Fig. 11.11. A better method of reading back a file, using EOF. This time, you don't have to know how many items are in the file.

place of FOR...NEXT. The condition in line 100 is EOF(1)=0, because while EOF(1)=0, we have not yet reached the end of the data for this file. In this particular short file, of course, the end is reached in the first batch of data. Try it out on the longer file of numbers!

Naming the names

Now that we have replayed and used a number file, it's time to start looking at some replaying methods for the file of names that we created earlier. When this file was created, each name was recorded, and the usual end-of-file marker would be placed on the tape. What we normally want to do is to place the names into an array, so that the computer can make use of the data. Using an array allows you to carry out tasks like placing the names into alphabetical order, for example. Not all uses call for an array, however. Suppose, for example, that you want to search the names for one beginning with the letter J. You could do this by using the program which is shown in Fig. 11.12.

The first few lines follow familiar patterns. When the program is run, it

```
10 CLS:PRINTTAB(13)"NAMEFINDER":PRINT
20 PRINT"This program will find a nam
e for ":PRINT"you from the NAMES file
."
30 PRINT"Press PLAY when the cassette
 is ":PRINT"ready, and then the space
bar."
40 IF INKEY$<>" "THEN 40
50 INPUT"First letter of name, please
 ";Q$
60 OPEN"CAS:NAMES"FOR INPUT AS 1
70 INPUT #1,N$
80 IF EOF(1)=0 AND Q$<>LEFT$(N$,1)THE
N 70
90 PRINT:PRINT"Name is ";N$
100 CLOSE:PRINT"Press STOP key on rec
order."
```

Fig. 11.12. Searching a file for one item, in this case, a name that starts with a given letter.

will allow you to find a name from the file simply by typing the first letter of the name. The NAMES file was recorded at the higher speed, but you *don't* have to use a SCREEN,,,2 statement in the replay program, because the computer adjusts itself. When the faster speed has been used, you must be sure that the cassette is fully wound back to just before the start of the file. The faster system is less tolerant of starting the playback at a point where the tones have already been recorded! Line 50 asks you for a first letter of a name (don't forget the capital letter!). Line 60 opens the file for reading, and line 70 starts a loop that takes a name from the buffer and checks first for the end-of-file character, and then for the first letter of the name being identical to the letter that you requested. If the whole file has not been read, this part compares the first letter that you have selected with the first letter of the name. If you have searched through the whole list without finding this letter, then line 80 will respond by ending the program when the end of file marker is found. The 'name' which is the last to be read in is X, which was used as a way of ending entry. This is then printed by line 90. If, on the other hand, the name that you want is found, then line 90 will print it, and the search ends also. You could, of course, have another test, so that if N$="X" then a message such as 'Name not on file' is printed in place of the 'X'.

This scheme works quite well with small amounts of data but only if all the data is different. If one name is Mary and another is Margaret, then a request for M will give you whichever of these comes first on the file. You will have to alter the tests in the loop if you want the program to print every name which starts with a given letter. Many programs of this type can be dealt with more satisfactorily by using an array to hold all the data in the memory. The cassette data system is then being used as a store only, and all of the selection is being done in the memory of the computer. You might wonder if there is any advantage here as compared to having the data in DATA lines. There is, because the data can be created by one program, and used by several others. You can make the program that reads and uses the data a fairly short one, so that there is room for a lot of data in the large memory of the MSX computer. The cassette data system, in other words, allows you to use short programs and lots of data.

Figure 11.13 shows how data from a program that has been put into the cassette data system can be read back into an array. If we are to read items into an array, we find ourselves facing a problem. The problem, you see, is that we have to dimension the array so that it will hold all of the items. To do this we need to know how many items there will be. It's easy enough if we used an array to hold the items when we recorded. Suppose, for example, that we INPUT the items into an array N$(J%), instead of recording directly. We could then open the file, record the value of J% with a PRINT#1,J%, and then set up a loop to record the array items. If we didn't use an array and didn't count the items at the time when we recorded, what can we do? One way out is to keep a note of the number of items, and enter it in response to a question in the replay program. For example, you could use:

```
INPUT"How many items ";N%:DIM NM$(N%)
```

to get your dimensioning. Another possibility is to use a counter in the recording program, such as:

```
INPUT NM$:N%=N%+1
```

and to record this number, using a filename that will relate it to the main program, on another piece of tape. For example, if the main program is filed as NAMES, the other one could be NUMBER. It may seem awkward, but it's a small price to pay for the sake of precise dimensioning. Precise dimensioning means that you will never get an error message because of an attempt to place too many items into an array. It also means that you are using the memory of the MSX computer in the most efficient way, and that can make the difference between being able to use as many items as you need and being restricted to a lot less.

If, however, you have a file like our existing NAMES file, in which the number of names is not known, perhaps because names are being added as the file is updated, then we have to take drastic steps. This is another respect in which disk filing is very much superior to cassette filing, because a number can be recorded on a disk either before or after a set of data, and the number read before the other data is needed. Figure 11.13 shows one approach which is not too time-consuming even for fairly long files. In this program, all of the items are read, one by one, and counted, until the end-of-file marker is found. You are then asked to rewind the cassette. The number of items in the file is now known, and an array can be correctly dimensioned. The file can then be read again, this time placing the items into an array. The

```
10 CLS:PRINTTAB(13)"NAMEFINDER":PRINT
20 PRINT:PRINTTAB(1)"Load the names f
ile as instructed."
30 PRINT"When the file is completely
loaded,":PRINT"type the first letter
of the names":PRINT"that you want to
see."
40 PRINT"Type 0 to stop the action."
50 PRINT:PRINT"Wind the NAMES cassett
e to the start.":PRINT"Zero the count
er."
60 PRINT"Press PLAY, then the SPACEBA
R when":PRINT"ready to start."
70 IF INKEY$<>" "THEN 70
80 PRINT:PRINTTAB(10)"PLEASE WAIT....
."
90 REM OPEN FILE
100 OPEN"CAS:NAMES"FOR INPUT AS 1
110 J%=0
120 INPUT #1,N$
130 J%=J%+1
140 IF EOF(1)=0 THEN 120
150 CLOSE:CLS:PRINT:PRINT"PRESS STOP
KEY"
160 PRINT:PRINT"Now rewind tape to st
art again, and":PRINT"press PLAY. Pre
ss SPACEBAR again":PRINT"when ready."
170 REM Load array
180 IF INKEY$<>" "THEN 180
190 PRINT:PRINTTAB(10)"PLEASE WAIT...
.."
200 OPEN"CAS:NAMES"FOR INPUT AS 1
210 DIM NM$(J%)
220 FOR N%=1 TO J%
230 INPUT #1,NM$(N%)
240 NEXT:CLOSE
250 PRINT:PRINT"PRESS STOP KEY NOW"
260 FOR X=1 TO 2000:NEXT
270 CLS:PRINT:PRINT"Please type first
 letter of name."
280 INPUT Q$:Q$=LEFT$(Q$,1):M%=0
290 FOR X%=1 TO J%
300 IF Q$=LEFT$(NM$(X%),1) THEN PRINT
NM$(X%):M%=1
310 NEXT
320 IF M%=0 AND Q$<>"0"THENPRINT"Cann
ot find the name.":PRINT"Please try a
nother one."
330 IF Q$<>"0"THEN 260
340 PRINT"END OF PROGRAM
```

Fig. 11.13. Reading a file into an array. The file is read once to find how to dimension the array, then again to get the items.

selection of names can then use a loop, because the file does not have to be read again.

Looking at the program in detail, lines 10 to 60 print instructions, and line 70 is the usual spacebar detector. In line 80, the 'Please wait' message is printed to remind you that you are waiting for the data to load. Variable J% is zeroed, and the fast speed is selected by the computer, because the original file was recorded at this speed. In the loop, each item is read, and the value of J% is increased for each item. The file must be closed in line 150, because we shall want to open it again later. We then have to rewind the cassette to the zero mark again, and messages are printed as a reminder. The value of J% is then used to dimension the array NM$ in line 210, just before the names are read from the file. Since we know the number of names, we can then use a FOR...NEXT loop in lines 220 to 240 to read the names into the array. Once the names are read, we can operate the NAMEFINDER action in lines 260 to 340. If a name is found, variable M% is set to 1. If the name cannot be found starting with the specified letter, M%=0, and the message in line 320 is printed. When you press the 0 key to end the finding action, this also causes M%=0, and the message in line 320 is suppressed by making the IF condition M%=0 AND Q$<>"0". If Q$="0", then the message is *not* printed. It looks a lot neater that way! The entry of "0" is detected also in line 330, and the program then does not loop back.

Making amendments

Let's be clear from the start that you cannot alter a file that is recorded on the cassette data system. What you can do, though, is to read in a file, make some alterations to it, and then re-record it. Using a cassette data system means that you can record the new file *using the same name* on another part of the cassette or on a different cassette. Using the same name allows the updated file to be read by the same programs. This is not so easy to arrange with a disk system. The technique which is shown in Fig. 11.14 follows the program of Fig. 11.13 very closely.

Lines 10 to 260 are virtually the same as the corresponding lines in Fig. 11.13, except for the instructions. These lines dimension an array, and then fill it with names. Line 150, however, now contains J%=J%−1. This avoids using the last item in the array, which is the terminator, X or 0, whichever was used in the program that created the file. In this way, when we extend the file we will have a continuous file with an X at the end, rather than a file which has these marks scattered through it at each place where the file had ended previously.

Lines 270 to 340 then re-record this file on a fresh piece of tape. You *can* record over an old file, but this is risky and, if you value all the effort that you have put into your files, don't do it! You are then asked to type more names. The stream is still open, so that no EOF mark has been recorded. This means

```
10 CLS:PRINTTAB(11)"FILE UPDATE":PRIN
T:SCREEN,,,2
20 PRINT:PRINTTAB(1)"Load the names f
ile as instructed."
30 PRINT"When the file is completely
loaded,":PRINT"follow the instruction
s about":PRINT"re-recording it."
40 PRINT"You can now add items to the
 file-":PRINT"Type 0 to end entry."
50 PRINT:PRINT"Wind the NAMES cassett
e to the start.":PRINT"Zero the count
er."
60 PRINT"Press PLAY, then the SPACEBA
R when":PRINT"ready to start."
70 IF INKEY$<>" "THEN 70
80 PRINT:PRINTTAB(10)"PLEASE WAIT....
."
90 REM OPEN FILE
100 OPEN"CAS:NAMES"FOR INPUT AS 1
110 J%=0
120 INPUT #1,N$
130 J%=J%+1
140 IF EOF(1)=0 THEN 120
150 CLOSE:CLS:PRINT:PRINT"PRESS STOP
KEY":J%=J%-1:REM Get rid of X
160 PRINT:PRINT"Now rewind tape to st
art again, and":PRINT"press PLAY. Pre
ss SPACEBAR again":PRINT"when ready."
170 REM Load array
180 IF INKEY$<>" "THEN 180
190 PRINT:PRINTTAB(10)"PLEASE WAIT...
.."
200 OPEN"CAS:NAMES"FOR INPUT AS 1
210 DIM NM$(J%)
220 FOR N%=1 TO J%
230 INPUT #1,NM$(N%)
240 NEXT:CLOSE
250 PRINT:PRINT"PRESS STOP KEY NOW"
260 FOR X=1 TO 2000:NEXT
270 CLS:PRINT:PRINT"PLEASE PREPARE TO
 RE-RECORD THE FILE"
280 PRINT"Find a clear space on the c
assette":PRINT"or use a new cassette.
 Zero the":PRINT"counter again and fo
llow"
290 PRINT"the instructions.":PRINT"Pr
ess REC and PLAY, then the ":PRINT"sp
acebar when you are ready"
300 REM Re-record data
310 IF INKEY$<>" "THEN 310
```

```
320 OPEN"CAS:NAMES"FOR OUTPUT AS 1
330 FOR N%=1 TO J%
340 PRINT#1,NM$(N%):NEXT
350 CLS:PRINT:PRINTTAB(14)"NEW ENTRY"
360 PRINT:PRINT"TYPE THE NAMES THAT Y
OU WANT TO ADD":PRINT"TO THE FILE NOW
. TYPE 0 TO END ENTRY":PRINT
370 INPUT"Name is- ";NM$
380 PRINT #1,NM$
390 IF NM$<>"X"THEN 370
400 CLOSE
410 PRINT"END OF PROGRAM"
```

Fig. 11.14. Updating a file. This has to be done by reading the file, making changes, and then recording again to a different part of the tape.

that anything else we add to the tape will be taken as part of the same file. While you are doing this, the buffer may fill, and names will be recorded. During this recording time, you will not be able to type more names. If you type only a few names, however, the buffer will not fill, and you will not hear the cassette system recording names until you have typed the X that ends the entry. This completes the new file. If you want to check it, then RUN again, and when you are asked to type more names, press STOP. Now type the line:

```
FOR Z=1 TO J%:?NM$(Z);" ";:NEXT
```

and ENTER. You will then see all of the names being listed on the screen. If you want to see the names listed in a more orderly way, or to select by letter, or to sort alphabetically, then you will have to write a piece of program for yourself! Note, incidentally, that we have done very little with MAXFILES. The reason is that we can only work with one file at a time when we use cassettes. The use of MAXFILES allows us to have several files open for reading or writing, but this is useful only in a disk system. Changing between reading and writing with a cassette means either changing cassettes or fast-winding to a new position, so there is no point in having files kept open to make it all slightly quicker for the computer!

Chapter Twelve

Editing, Fault-Tracing and Miscellany

In a computer which permits as many commands as the MSX computer, it's impossible in a book of this size to cover everything in detail. As your experience and confidence grow, however, you'll find that you learn new tricks faster, and that there are some instructions which only start to look useful when you have come up against a problem that can't be solved easily any other way. In this chapter, we'll be looking at a lot of the instructions that can be used for making your programming easier, or for tracing faults, or which just didn't fit anywhere else.

To start with, there is the AUTO command. A lot of published programs are written with line numbers that ascend in tens. If you type AUTO (then RETURN), or just press the F2 key, then the word auto appears on the screen – and so does the line number 10. You can then type your line 10, and when you press RETURN, the next line number, 20, will appear. Using this scheme, you don't have to type any line numbers, which is good news if you find that you tend to forget them. Suppose you want to start at line 100? No problem: just type AUTO 100, then RETURN, and your first line number will be 100. You can still use the F2 key to get the word auto. You want lines that go up in fives, not tens? Then use AUTO 50,5 and you find that your first line is 50, the next is 55 and so on. It's a very useful dodge when you have a lot of lines to enter. At the end of the program, just press CTRL/STOP, and the automatic line numbering will stop.

As well as entering lines automatically, it's often useful to be able to delete lines automatically. If, for example, you have a program with a useful subroutine which you want to save on tape, it's useful to be able to delete lines 10 to 4500, and be left with the subroutine which uses lines 5000 to 5050. There are computers, believe it or not, on which you have to type each line number, then RETURN, just to do this! On the MSX computer, you type DELETE 10-4500, press RETURN, and it's done. You might wonder how anyone could manage without it. The answer is – not very well!

There's another command which is closely related to these. Suppose that you have a program which is neatly numbered in tens, and that you remove three lines, 240 to 260 inclusive. This might be because you have been able to use a subroutine instead of these three lines. Your program now has a hole in it. Even more likely is that you have to add lines like 126, 127 to a program just to get something extra in. The MSX computer allows you to make everything neat again by using the RENUM command to renumber your lines. Using RENUM (then RETURN) will number your program starting at 10 and numbering in tens. RENUM100,10,5 will start your line numbers at 100, and number in fives. You can renumber just as you want to, and you don't have to renumber all of a program. If, for example, you type RENUM2000,160,10, then press RETURN, you'll find that your program is as it was as far as line 150, but the next line is 2000, and it's numbered in tens from then on. There are computers which can't do this. Makes you wonder what people can do with them.

Now what do you do with a subroutine when you want to use it in another program? One useful answer is to save it on tape, but there is a special and rather useful command for this. Instead of using CSAVE as you would for a complete program, use the command SAVE. You have to follow this with a filename, like "CAS:mysub", placed within quotes, and you'll then have to press PLAY and REC on the recorder before pressing RETURN. Record the subroutine, and then wind back the tape. Now use NEW to clear the machine, and then type a few lines which have lower line numbers than the lines of your subroutine. For example, if your subroutine has lines 5000 to 6000, then type lines 10 to 100. Now type MERGE "CAS:mysub", using the filename that you picked for your subroutine. Press RETURN, and then press the PLAY key of the recorder. You will see the 'Found:mysub' message when the subroutine is found, and soon after, the O.k. appears. Now list, and you will see that you have a merged program. Your lines 10 to 100 are now joined with lines 5000 to 6000 of the subroutine. This sort of thing, which is not available on all computers, even some at fancy prices, encourages you to program in the way that I have outlined in Chapter 6, using subroutines. You can keep a stock of useful subroutines on tape, ready to use. I find that there are subroutines that I use in all of my programs. There's always a Press-any-key routine, usually some sound effects, often something that sorts words into alphabetical order. You, too, will find that you have favourite subroutines, and you'll greatly treasure this useful set of commands. One thing you need to watch, though, is that anything you have saved by using SAVE must be loaded by using MERGE or LOAD, *not* CLOAD.

Editing

Editing means changing something that has already appeared on the screen. Any feature of a line, including its line number, can be changed by editing. The editing process can be carried out

(a) while a line is being entered, before RETURN has been pressed;

(b) at a later stage, after a line has been entered, but before the program is run;
(c) when an error is signalled during running.

For any form of editing, the line that you want to edit must be visible on the screen. It doesn't matter whereabouts on the screen it appears, or what other lines or commands are placed around it. *If you can see it, you can edit it!*
Dealing with these in order:

(a) While a line is being entered, all of the editing commands, below, can be used. Editing is completed by pressing the RETURN key. This works even if you are using auto line numbering.
(b) When the line has been entered, but the program has not been run, you must make sure that the line is on the screen. If it is not, then type LIST xxx, where xxx is the line number, and press RETURN. Remember that the key F4 gives you the word 'LIST', and key F9 (SHIFT F5) gives 'LIST.'. The difference is that LIST. will place the *current line* on the screen. This is the line that you have just entered. After editing, you should then be careful to use the cursor down-arrow key to move the cursor below the last line on the program before using LIST or RUN.
(c) When the program stops with an error message, the number of the line in which an error has been traced will be put on to the screen. This does not always mean that there is an error in *this* line. For example, if the line contains READ X, and only strings can be read, an error will be signalled in this line, because this is where the READ takes place. You need to alter a RESTORE or a DATA line.

Editing commands

(1) *Cursor movement.* The cursor arrowed keys of the keyboard of the MSX computer allow you to move the cursor around the screen. Many MSX machines group these keys very conveniently together at the right-hand side of the keyboard.
(2) *To replace a letter*, simply place the cursor over the error and type the correct letter.
(3) *To delete a letter*, place the cursor over it and press the DEL key.
(4) *To insert a letter*, place the cursor over the letter that will follow the insertion. Press the INS key, and then type the letter. You can then make several insertions without pressing INS again. Pressing INS again stops insertion. Insertion is also stopped when you press the cursor controls or the RETURN key. When the cursor is being used for insertion, it changes to half its normal height. Watch out for this, because it's a good way of checking that insertion will actually take place. It's annoying to think that you are inserting when you are, in fact, *replacing* letters!
(5) *Pressing CTRL E* (CTRL key and E key together) will delete everything

in a line to the right of the cursor. 'Line' in this sense means a complete numbered BASIC line, which may take several lines on the screen.
(6) *The CTRL I key pair* moves the cursor eight spaces to the right. You can also use the TAB key to do this.
(7) *Pressing RETURN* ends editing, and places the line into memory. If you don't press RETURN after editing a line, but only move to another line by using the cursor controls, then *no editing is carried out.*

Figure 12.1 lists the uses of the CTRL and other keys pressed together. Many of these actions duplicate others. You can obtain these effects in a

KEY=key which is pressed along with CTRL.

KEY	Effect
A	Make next character a graphic.
B	Move cursor to first character of word to the left.
C	Stop program.
E	Delete rest of line.
F	Move cursor to first character of next word.
G	Sound beep.
H	Backspace cursor by one step and delete character.
I	Move cursor eight spaces to the right.
J	Move cursor to first position on next line.
K	Move cursor to top left corner of screen.
L	Clear the screen.
M	As for RETURN key.
N	Cursor to end of line.
R	Insert character at cursor position.
U	Delete line.
\	Cursor right.
]	Cursor left.
^	Cursor up.
SHIFT-	Cursor down.

Fig. 12.1. The actions that you can obtain by using the CTRL key along with others.

program by using an instruction such as PRINT CHR$(11), which homes the cursor to the top left-hand side of the screen.

Digging out the bugs

In computing language, a fault in a program is called a *bug*, and someone who puts the faults there is called, of course, a programmer. Your programs can exhibit many kinds of bugs, and a lot of these are indicated by the error messages that you get when you try to run a program. Some of these messages are pretty obvious. 'Undefined line number', for example, means that you have used a command like GOTO1000 or GOSUB1000 and forgotten to write line 1000. It can also appear if you have tried to DELETE1000 with no line 1000, or if you have a THEN1000ELSE2000 following an IF somewhere.

The most common fault message is 'Syntax error'. This means that you have wrongly used some of the reserved words of BASIC. You might have spelled a word incorrectly, like PRIBT instead of PRINT. You might have missed out a bracket, a comma, a semicolon, or put a semicolon in place of a colon. Machines can't tell what you meant to do, they can only slavishly do exactly what you tell them. If you haven't used BASIC in exactly the way the machine expects, you'll find a syntax error being reported. Another common error is 'Illegal function call'. This usually means that something silly has happened involving a number. You might, for example, have used TAB(300). Of course, having read this book, you wouldn't write TAB(300) in a program, but you might have TAB(N), and the value of N has gone to 300 in some sneaky way. Anything that makes use of numbers, like MID$, LEFT$, RIGHT$, INSTR, STRING$, and others can have an incorrect number used – and this will cause the 'Illegal function call' error. You will also find that using a negative number in SQR(N), a negative or zero value in LOG(N), and other mathematical impossibilities will cause this error message. The cause shouldn't be hard to trace, because the machine tells you which line caused the trouble.

A lot of errors can find their way into programs, even when you are entering a program that has been printed in a magazine. In general, the programs that you find in the monthly computing magazines are pretty reliable, but some are printed in a way that makes it difficult for you to enter them correctly. The main problems arise when the author of the program has used I (capital I) or l (small L) as a variable name, or has used a printer which does not have slashed zeros. Of these, confusion between O and 0 is the worst. A line like:

```
IFM=10ORJ=4ORD=2
```

can cause a lot of trouble, and one magazine seems to specialise in lines like this! If it had been printed as:

```
IF M=10 OR J=4 OR D=2
```

all would have been clear. Sometimes this has to be done just so as to be able to get a program to fit into the memory of a computer, but this is the only real excuse. You have to be particularly careful with sound programs, because of the use of O to mean Octave. In your own programs, beware of using letters O and I as variable names.

Even when you have eliminated all of the syntax errors and illegal function calls, you may still find that your program does not do what it should. The MSX computer does what any machine of the nineteen eighties should do – it gives you a lot of ways of finding out exactly what has gone wrong. One of the most powerful of these is the STOP key. This, as you know, stops the action of the machine when you press it, and restarts it when you press STOP for a second time. This, as we'll see later, can be very useful for graphics bug-hunting, but for other programs, pressing CTRL and STOP is more useful. This stops the program, and prints the line number in which the program stopped. What you probably don't know, however, is that you can print out the values of variables, and even alter values while the program is stopped, and then you can make the program resume by using CONT. Suppose, for example, you try the simple program in Fig. 12.2. This is a slow count, and you should run it, and then press CTRL STOP at some early stage. The program stops, and you get a message like 'Break in 30'.

```
10 FOR N=1 TO1000
20 PRINTN
30 FORJ=1 TO 500:NEXT
40 NEXT
```

Fig. 12.2. Illustrating the use of the CTRL/STOP method of checking.

That line number, 30, is important, because this is where the program has stopped. You can start the program again at that line, using CONT, *providing* you don't edit, delete or add to any of the lines of the program. Try typing ?N,J and RETURN. This will give you the value of N and J. Now try N=998 and press RETURN. Type CONT, then RETURN. You will then see the count start again – but at 999! This is an excellent way of testing what will happen at the end of a long loop. Testing would be a rather time-consuming business if you had to wait until the count got there by itself. You can even make this testing process automatic! Take a look at Fig. 12.3. This

```
5 STOP ON:ON STOP GOSUB 1000
10 FOR N=1 TO1000
20 PRINTN
30 FORJ=1 TO 500:NEXT
40 NEXT
100 END
1000 PRINTN,J:N=998:RETURN
```

Fig. 12.3. Automatic checking with CTRL/STOP, using a subroutine.

uses line 5 to make sure that a subroutine is run whenever the CTRL and STOP keys are pressed together. In this case, the effect will be to print the values of N and J, alter the value of N to 998, and then continue. To cancel the effect of line 5, you can add the line:

50 STOP OFF

STOP used alone can be most useful when you have a graphics program that has gone wrong. When you press STOP, the graphics action will be frozen, and you can use this to see in what order things happen. Pressing STOP again resumes the action, and there is no limit to the number of times that you can press the STOP key to check on how the picture is changing. If this alone isn't enough, add a delay loop temporarily to your graphics program, and run it in slow motion, using STOP to check the tricky parts. The alternative is to put in an ON STOP GOSUB routine which includes a time delay.

Tracing the loops

One way in which a program can be baffling is when it runs without producing any error messages – but doesn't run correctly. This is really a sign of faulty planning, but sometimes it's an oversight. The ON STOP GOSUB method of tracing a fault can then be very useful, because it allows you to print out the state of the variables at any stage in the program, and then carry on. Sometimes you want a simpler form of tracing, though. If your program contains a lot of IF...THEN...ELSE lines, it often happens that one of these does not do what you expect. In such a case, the MSX computer provides help for you in the form of two commands TRON and TROFF.

TRON means TRACE ON, and its effect is to print on the screen the line number of each line as it is executed. The line numbers are put between square brackets, and they are printed at the start of a screen line, in front of anything the program prints. Try printing TRON and then running the program of Fig. 12.3. TRON is particularly useful if you aren't sure what a program is doing, and it can be very handy in pointing out when something goes wrong with a loop.

Remember that you can combine TRON with other de-bugging commands. You can, for example, stop the program, alter the variables, and then continue, with TRON showing you which lines are being executed. Typing TROFF (then RETURN) switches off this tracing process.

Error trapping

Earlier in this book, we came across the idea of mugtrapping. This is a way of checking data that has been entered at the keyboard, to see if it makes sense or not. The mugtrapping is carried out by using lines such as:

```
60 IF LEN(A$)=0 THEN GOSUB 1000:GOTO 50
```

and you need a separate type of mugtrap for each possible error. This can be fairly tedious, and it usually turns out that there is one other error that you haven't spotted. The MSX computer is one of the exclusive few machines that offer you another mugtrapping command, ON ERROR GOTO or ON ERROR GOSUB. Figure 12.4 gives a very artificial example – a real-life

```
10 ON ERROR GOTO1000
20 PRINT"Type a word please"
30 INPUT A$
40 L=LEN(A$)
50 PRINT1/L
60 END
1000 PRINT"Word has no letters!"
1010 RESUME 20
```

Fig. 12.4. Using the ON ERROR GOTO command.

example would involve too much typing. In this example, the length of a word is measured, and the number is inverted (divided into 1). This is impossible if the length is zero, and the ON ERROR GOTO is designed to trap this. You could get a zero entry, for example, by pressing RETURN without having pressed any other keys. Now normally, when this happened, you would get an error message, and the program would stop. The great value of using ON ERROR GOTO is that the program does not stop when an error is found; instead it goes to the subroutine. In this example, the subroutine prints a message, then resumes on line 20. Using RESUME by itself would cause the program to go *back* to the line which contains the error. Unless the subroutine has corrected the error, this can cause an endless loop! Error trapping is delightfully simple, but it's something that calls for experience. You see, if your program still contains things like syntax errors, these also will cause the subroutine to run, and this can make the program look rather baffling as it suddenly goes to another line. You can use RESUME NEXT if you want the program to skip a line, and this is often more useful, because it does not take you back to a fixed line number like 20. If you use RESUME NEXT, then an error in line 100 will take you to 110, and error in line 540 will take you to 550, and so on, assuming that your lines are numbered in tens.

Figure 12.5 shows another example. Line 20 ensures that any error will

```
10 CLS
20 ON ERROR GOTO 1000
30 FOR N%=1 TO 5
40 READ X%:Y$=STR$(CSNG(SQR(X%)))
50 PRINT"Number ";X%;" ";"square root
";Y$
60 NEXT
70 DATA 5,4,3,-2,2
80 END
1000 Y$=STR$(CSNG(SQR(ABS(X%))))+"J"
1010 RESUME NEXT
```

Fig. 12.5. Another example of error checking, with an automatic resumption.

also by using CHR$(13), and this is often a useful addition to LIST. You can use up to fifteen characters on each of these keys, and by careful selection, you can save yourself a lot of typing. It's useful, for example, to have PRINTTAB(on a key, and SOUND can be another useful word in some programs.

Finally, one odd item. If you want to control the motor of the cassette recorder separately, you can use the commands MOTOR ON and MOTOR OFF. This allows the computer to control background music for colour displays, or spoken instructions for educational programs when the jack plug is removed from the EAR socket. You can also use this to allow the fast forward and rewind actions to take place without removing the motor control plug. This can be useful in many of the programs in Chapter 11 which require the cassette to be rewound.

That's the end of this particular road for me, but only the beginning for you. Any model of MSX computer is a fascinating bag of tricks, all of which you will gradually learn to unlock. This book should have unlocked some of the secrets for you, and the rest is now up to you. By this time, you should be able to cope with the way in which commands are described in the manual. From now on, each time you settle down to write a program, you will be learning more about your computer. This is something that you can never acquire from running programs, or even from studying programs that were written by other people. By all means, look carefully at programs in the magazines. You will find that a lot of programs that are written for the TRS-80 or for the Dragon can be adapted for the MSX computer, and so also can programs for the Colour Genie and Spectravideo SV-318 and SV-328. Very soon, you will find a huge variety of program ideas which are available for you to use. Happy programming!

Appendix A
TV and Cassette Hints

The set-up and connection of your TV receiver and cassette recorder to the MSX computer should be well dealt with in the manual that comes with the computer. There are a number of useful hints, however, particularly on tuning TV receivers and checking and adjusting cassette recorders that are not dealt with in the manuals. This Appendix is a guide which will take over where the manual leaves off.

One of the first things that you may need to attend to is an aerial splitter. If you use one TV for entertainment and for computing, it does it no good at all to have to keep plugging and unplugging aerial connections. The answer is a two-way aerial splitter, such as the one illustrated in Fig. A.1. This is sold under various names, such as *Pandapack*. When you have this plugged in to the TV, the TV aerial lead can be plugged into one socket and the lead from the computer into another. All you need to do to switch from *Dallas* to computing is to switch on the computer and change channels! If you can keep a second TV for use with the computer, of course, then you won't need this.

The alternative to a TV receiver is to use a monitor. A monitor looks like a TV, but it can't receive signals from an aerial. It produces a much clearer

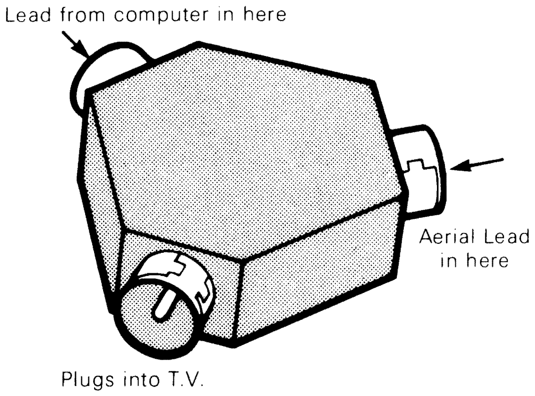


Fig. A.1. A two-way splitter for the aerial socket of your TV. Using this, you can keep the aerial connected, and also have the computer lead connected permanently.

picture, and doesn't need tuning, because it uses the signals from the computer directly. Some types of monitors can also be used along with Video Recorders so as to get a clearer picture from recordings also. The MSX computers can be connected to a monitor by using the socket that is marked VIDEO – the type of socket and the position of it will vary from one make of computer to another. A special connecting cable will also be required. The monitor *must* be one that uses a composite video input.

It's a good idea to see how many mains sockets you have around, and where you are going to house everything. When you are in full control of your computer you will need three mains sockets, one for the computer, one for the cassette recorder, and one for the TV receiver. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of a three- or four-way socket strip with a cable and a plug (Fig. A.2). This avoids a lot of clutter – you don't want to bring your computer crashing to the floor when you trip over a cable. Don't rely on the old-fashioned type of three-way adaptor – they never produce really reliable contacts. Most of the MSX computers have their own on/off switches, but you should *always* take out the mains plug when you have finished a computing session.

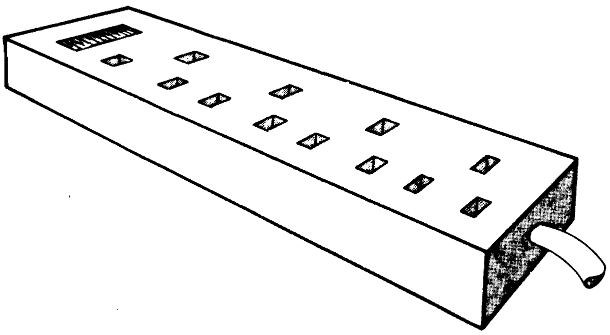


Fig. A.2. A four-way socket strip which avoids the use of the old-style adaptors.

When you have the essential basic equipment, consisting of the computer keyboard, TV or monitor and the cassette recorder, quite a lot of flat surface is needed. Later on, you will probably want to add joysticks, a printer, disk drives and other extras which make the difference between having a real computer system and just having a computer. All of this needs space, and the best way that I have found of organising this is one of the computer stands made by Selmor (Fig. A.3). If you aren't at that stage yet, then a good-sized desk or table will have to suffice for the time being. Computing is like hi-fi – there's always something else that you can buy!

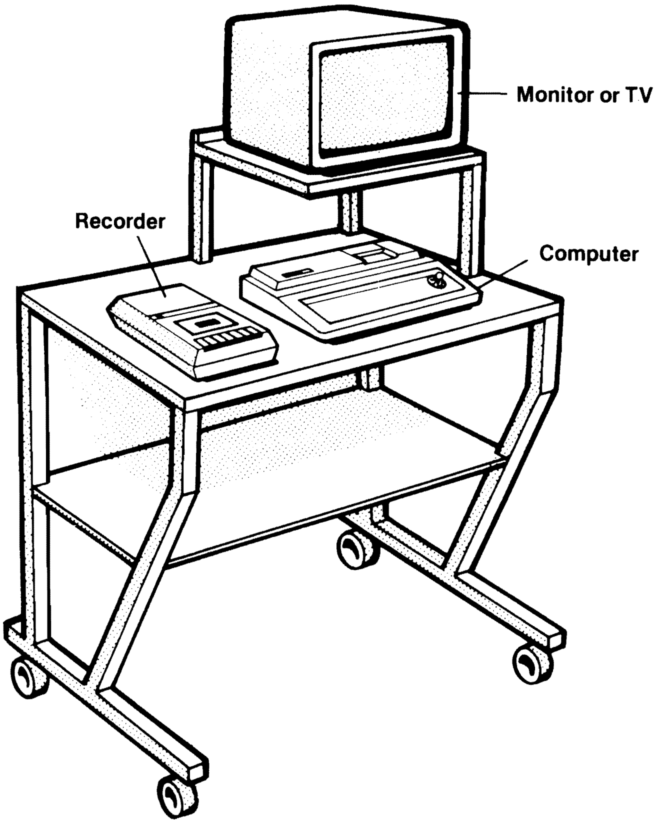


Fig. A.3. Using a Selmor stand to house all the bits and pieces of a typical computer system.

Tuning a TV receiver

Unless you are exceptionally lucky, you will probably see nothing appear on the TV screen, and hear only a loud rushing sound from the loudspeaker of the TV when you first try it. This is because a TV receiver has to be tuned to the signal from the MSX computer. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked VCR it's unlikely that you will be able to get the MSX computer tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the MSX computer's signals.

Figure A.4 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. A.4(a). This is the type of tuning system that you find on black/white portables, and to get the MSX computer's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial

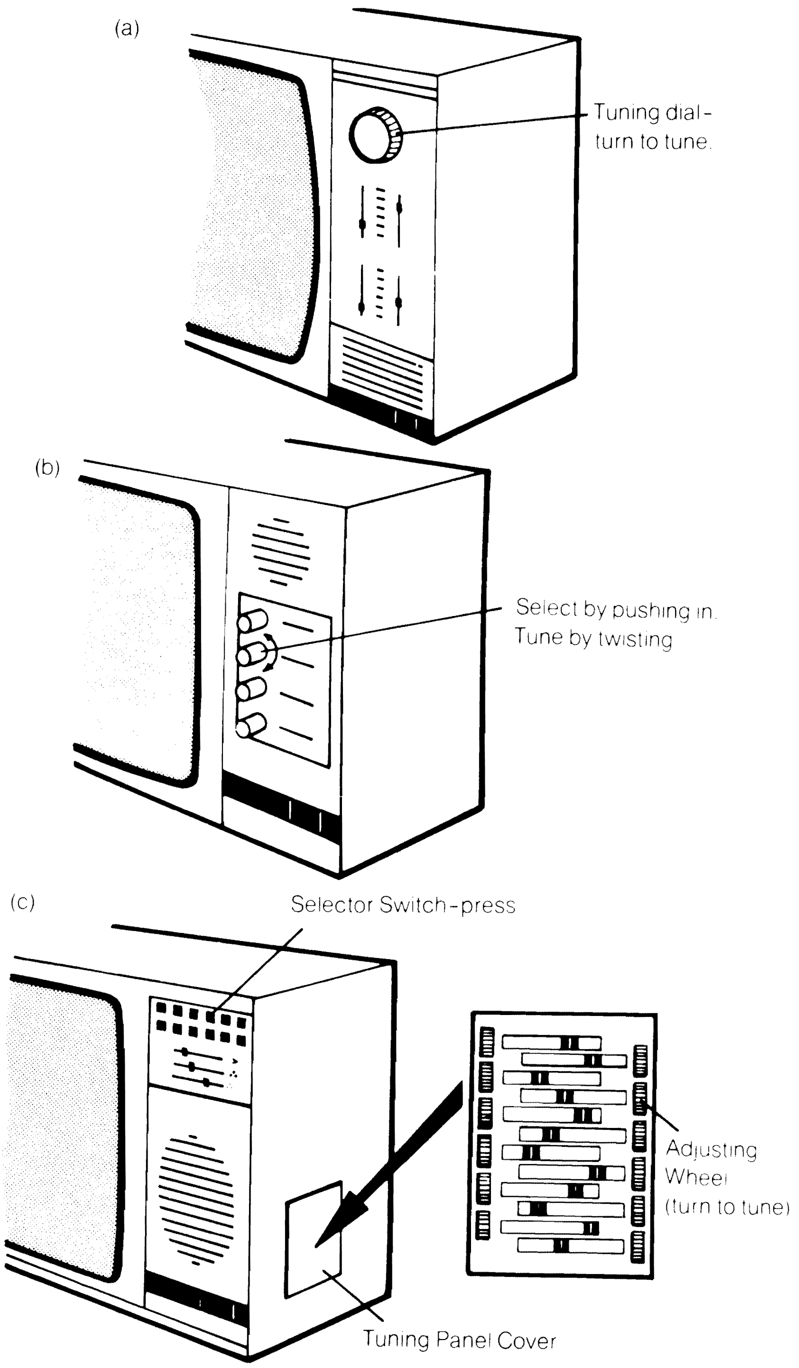
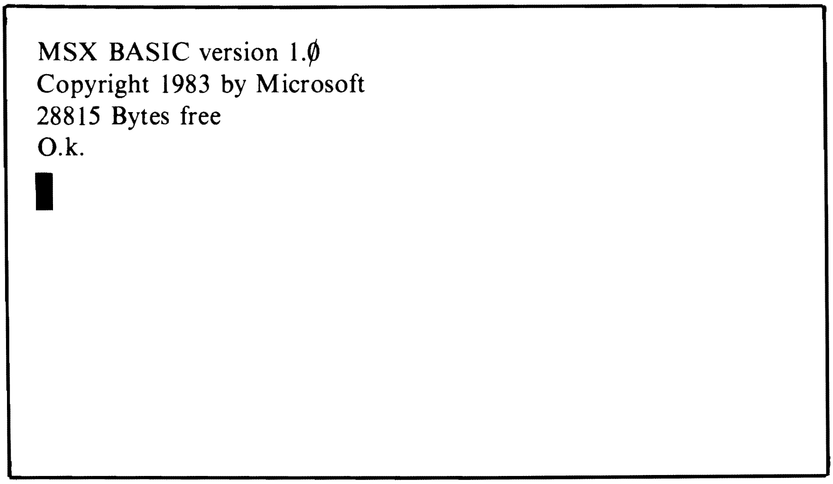


Fig. A.4. TV tuning controls: (a) single dial, as used on B&W portables, (b) the four-button, (c) the more modern touch-pad or miniature switch type.

isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the MSX computer signal appear. If you turn the volume control up slightly so that you can hear the rushing noise of the untuned receiver, you will hear things go quiet as the MSX computer signal appears. You may find that there is some reduction in the sound level as you tune to a local TV transmission, but you'll notice the difference. The MSX computer doesn't give you the sound of *Dallas*!

What you are looking for, if the MSX computer hasn't been touched since you switched it on, is the screen display that is illustrated in Fig. A.5. The top part of this display is a copyright notice, along with a note of how much memory is free for you to use.



```
MSX BASIC version 1.Ø
Copyright 1983 by Microsoft
28815 Bytes free
O.k.
█
```

Note: The number of bytes free will vary, depending on which machine is in use.

Fig. A.5. The copyright notice which should appear on your screen when the tuning is correct.

When you can see the words as in Fig. A.5, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. If you turn up the volume control setting, you should find that the amount of rushing sound that you hear is at its lowest, almost silent. On a colour TV receiver the words may never be particularly clear, but get them steady, at least, and as clear as possible.

The older types of colour and B/W TV receivers used mechanical push-buttons, as shown in Fig. A.4(b), which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one – which for most of us means the fourth one. Push this one in fully. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the MSX computer's signal during this time, you'll see and hear the same signs – the message on the screen and the reduction in the noise from the loudspeaker. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the MSX computer signal at any setting, check that the plugs on the TV aerial cable are not loose. If you have an aerial handy, plug it in and use the other tuning buttons to check that you can receive normal TV signals. If you can, there's nothing wrong with the TV, so switch back and try again to find the MSX computer signal.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver, as in Fig. A.4(c). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen and silence from the loudspeaker. On this type of receiver, the picture is usually fine-tuned automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep retuning. The MSX computer should give a good picture on practically any TV receiver. I tried it with several, and even my Philips portable colour TV, which does not work well with computers, gave a reasonably good picture with the MSX computer. If your TV exhibits faults like a shaking picture, or very blurred colours, then check the tuning carefully. If the faults persist, and the TV is correctly tuned, you will have to contact the service agents for the TV – or use a different model in future!

The cassette recorder

The computer has circuits which will convert the instructions of a program into musical tones, which can then be recorded on any cassette recorder. When these notes are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of a cassette recorder allows you to record your program on tape and to replay them again. If you don't already have a cassette recorder, buy one which has a tape counter and which allows motor on/off control, because this is much more suited to the MSX computer. You don't need a special hi-fi model, a reasonable make of portable battery/mains machine is ideal, and there are lots to choose from. You could, for example, buy one which is manufactured by the same makers as your computer. If you already have a recorder, you can probably use it, but life will be harder if it does not allow for remote control of the motor, and has no tape counter.

The next thing that you have to sort out is a supply of blank cassettes. There's nothing wrong with using reputable brands of C90-length cassettes (ordinary ferric tape, not the hi-fi CrO_2 type), but you'll find that the short lengths of tape that are sold as C5, C10 or C15 in computer shops and in most branches of W. H. Smiths, Boots, and Currys are much more useful. Then make sure that the recorder is connected as shown in your manual. Put a fresh cassette into the machine, with the 1 or A side uppermost. The first part of the cassette tape consists of a *leader*, which is plain, not recording, tape. This has to be wound on before you can record. Reset the tape counter to zero by pressing the little button, and then fast-wind the cassette to a count of 5. Now before you can make a recording to test the system, you need a program to record, and this involves some typing. If you have no program in the machine, then type four lines of REM, as in Fig. A.6.

```
10 REM
20 REM
30 REM
40 REM
```

Fig. A.6. A program for testing the cassette recording and replaying actions

Now make sure that the cassette recorder is ready, with the cassette in place and wound on to a count of 5. Type CSAVE"test", and then press the PLAY and RECord keys on the tape recorder. Press them firmly so that they lock in place. The motor of the recorder should not start running until you press RETURN, unless you have no remote motor control. After a short time, the 'O.k.' and the cursor of the MSX computer will reappear on the screen, and the motor of the recorder will stop automatically. This lets you know that the program has been recorded, and you can press the STOP key of the recorder so that the tape is released. Get into the habit of pressing the STOP key after completing a recording, because if you don't, it can damage the rubber wheel that moves the tape. That's all that's involved in making the recording. Now comes the crunch. You have to be sure that the recording was successful. Wind back the tape again. Type NEW and press RETURN. This should have wiped your program from the memory.

You can now load the instructions in from the tape. Type either CLOAD or CLOAD"TEST" and press RETURN. Now press the PLAY key of the recorder (did you rewind the tape?). As the tape plays, you will first see the message: 'FOUND: test' appear on the screen, assuming that you used the filename of test when you recorded. This will be followed very quickly by the 'O.k.' message to show that the loading operation is complete. When this appears, the program is in place, and the recorder motor stops. You should then press the STOP key of the recorder. Type LIST now, then press the RETURN key. You should see your program appear on the screen.

Cleaning tapeheads

Any cassette recorder which gets a lot of use will eventually need head cleaning. The tapehead is the part of the machine that the tape rubs against, and any dirt on the tape will accumulate on the head. This has the result of building up a film of dirt between the head and the tape. This makes the signals from the tape fainter and more distorted when we replay tapes, and can result in errors appearing in programs. Fortunately, it's simple to clean a head, using one of the head-cleaning kits that you can buy in Boots, W. H. Smiths, or any audio dealers. These work very much more satisfactorily than the cleaning-tapes that you can also buy.

The cleaning kit contains some liquid, some cloth, and usually some pads on the end of plastic rods. Be careful not to spill the cleaning liquid over your notes, because it will cause anything written in ball-point ink to smear, and it might dissolve some types of plastics. Start by switching on the computer and opening the recorder lid, as if you were going to put a cassette into it. Now press the PLAY key. When you do this, you will see the recording head appear. This is the head in the middle of the gap, not the (plastic) one at the left-hand side. There will be a rubber wheel at the right-hand side. Moisten a cleaning pad with some of the cleaning liquid, and rub the moist pad against the curved face of the recording head for a few seconds. Take another clean pad, moisten it with cleaning liquid, and hold it gently against the rubber wheel as it revolves. Don't put so much pressure on the wheel that it stops. Leave the recorder with the lid open for a minute so that the liquid can evaporate, then press the STOP key, and shut the lid again. Once every few months should be often enough for this unless you are using cassettes for hours each day.

Finally, if you find that your recorder works perfectly with your own tapes, but refuses to load commercial tapes, or tapes belonging to your friends, then the fault is tapehead alignment. You'll find advice on this in Appendix B, which follows.

Appendix B

Cassette Head Adjustment

Cassette recorders, like open-reel tape recorders, work on the principle of pulling plastic tape, which has been coated with magnetic material, past a 'tapehead', which is a miniature electromagnet. The important part of any tapehead is the 'gap', a tiny slit in the metal, too fine to see except under a microscope. This slit should be placed so that it is at 90° to the direction of movement of the tape, but this angle, which can be adjusted by tilting the whole tapehead, is seldom precisely set, even when the recorder has been quite expensive. A poorly set-up head will make it difficult to load programs that have been recorded on correctly set-up equipment (bought software, for example), though you will always be able to load tapes which have been

CASSETTE RECORDER HEAD ALIGNMENT METHOD

(1) Insert a cassette, with a long program, into the recorder.
(2) Remove cable connections between the computer and the recorder.
(3) Start playing the cassette. Set the volume control to a comfortable level, and listen. Any tone control should be set to give maximum treble.
(4) Insert a thin-bladed screwdriver into the head-alignment screw-head. On some recorders this is reached with the cassette flap shut, through a hole in the casing. On other models, it will be necessary to open the flap. This may have to be done *before* playing the tape.
(5) Adjust the azimuth screw *slightly* in each direction, listening to increase in the treble (a sharper sound). If adjustment causes the note to sound more muffled, reverse the direction of turning. Adjust until the note is at its sharpest.
(6) Rewind the cassette, and make the connections between the computer and the recorder.
(7) Try to load a program. If good loading cannot be achieved, repeat the procedure, but look for *another* setting which produces maximum treble.
(8) NOTE that this procedure is needed only if a tape from a reputable source cannot be loaded. Tapes made on a recorder will be loaded by that recorder unless there is a serious fault. Once the adjustment described above has been carried out, tapes recorded *before* the adjustment may not load correctly *after* the adjustment.

saved on the same equipment with the same head adjustment. NEVER touch the recording head with anything metal – but you can set the alignment fairly easily, following the scheme outlined here, in Fig. B.1.

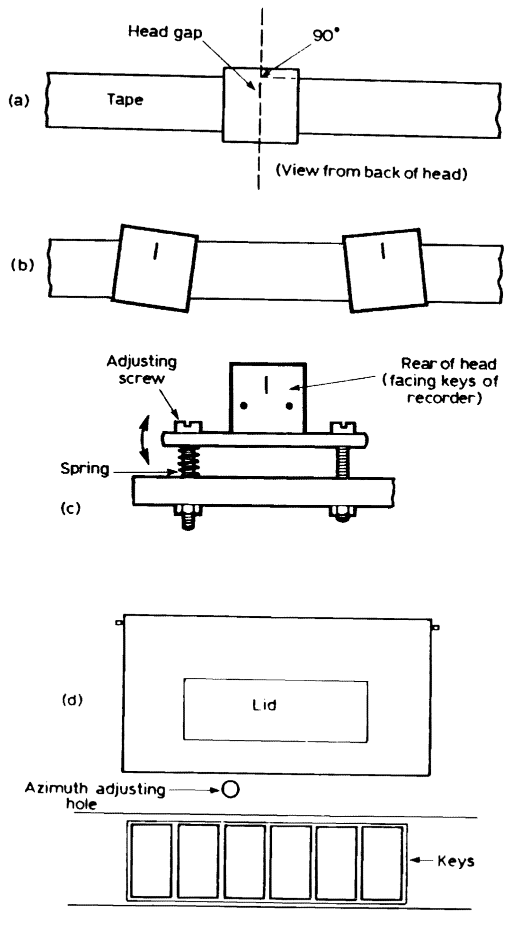


Fig. B1. Tape-head azimuth. The narrow slit in the tape-head (a) is normally at 90° to the edge of the tape. This is the correct azimuth angle, but a surprising number of recorders have this maladjusted. Any deviation from this angle (b) causes muffled sound and poor loading. The angle can be altered by turning an adjusting screw (c) which is on the head mounting. This is often reached through a hole in the casing of the recorder (d). (Courtesy of Keith Dickson Publishing.)

Appendix C
Some Other Commands

Even in a book of this size, it's impossible to deal with all of the commands of MSX BASIC. One reason is that there are quite a number of commands that require a knowledge of much more than BASIC to make use of. Several commands, for example, are provided so as to make the machine very useful for machine code programmers. Among these are BLOAD and BSAVE. These are versions of CLOAD and CSAVE which load to and save from specified parts of the memory. These commands will, for example, allow you to save machine code programs separately from BASIC, and also to load a machine code program without disturbing a BASIC program which is already in the memory. DEF USR is used to put into a BASIC program reference address numbers for machine code programs, so that the BASIC program may call these up by means of the USR command. This is because many subroutines which take a long time to run in BASIC will run very fast in machine code. CALL is used to run routines from a ROM cartridge. BASE is used to find where video patterns are stored, and VARPTR is used to find where variable values are stored. VPEEK and VPOKE are used in connection with the video display memory (try VPOKE120,65, for example), and PEEK and POKE will affect the main memory. VDP is used to alter values in the video display registers (like the sound registers). All of these commands require you to have a good knowledge of what goes on inside the computer. If you program only in BASIC, it's most unlikely that you will ever need any of them.

Other commands have been omitted because they are seldom needed when you first start to program. Of these, LPRINT and LLIST refer to printer use, and should not be used unless you have a printer on line. They operate in the same way as the familiar PRINT and LIST. LPOS is a way of finding the position of the printer head. One of the SCREEN commands can be used to switch between MSX and non-MSX printer types. Of the commands which relate to joysticks, only STRIG has been mentioned, because not many readers are likely to be *writing* programs which use joysticks right away. Similarly, I have omitted some of the less useful sound commands such as the use of PLAY to find if a string is being played or not. Life's too short for all of them, and by the time you are ready for them, you will have found out how to make effective use of the definitions of keywords in the manual.

Index

MSX BASIC is the most versatile and extensive version of BASIC yet released. It's also the first attempt to standardise home computers, and is attracting keen international attention.

This book provides a clear and complete step-by-step course, accompanied by a very large number of practical examples, and takes the user through the whole of the MSX BASIC repertoire. It has been designed to appeal not just to beginners or newcomers to Microsoft types of BASIC: it will also deal with more advanced commands and topics.

The Author
Ian Sinclair has contributed regularly to journals such as *Personal Computer World, Computing Today, Electronics and Computing Monthly, Hobby Electronics* and *Electronics Today International*. He has written over fifty books on aspects of electronics and computing, mainly aimed at the beginner.

Front cover illustration by Godfrey Dowson

COLLINS
Printed in Great Britain
0 00 383103 5